

UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Technology
Robotics and Computer Engineering curriculum

Gautier Reynes

VR-Enhanced Remote Inspection Framework for Semi-Autonomous Robot Fleet

Master's Thesis (30 ECTS)

Supervisor(s): Robert Valner, PhD
Ulrich Norbistrath, PhD

Tartu 2023

VR-Enhanced Remote Inspection Framework for Semi-Autonomous Robot Fleet

Abstract:

This thesis presents the design and development of a Virtual Reality (VR)-enhanced user interface and communication infrastructure for remote inspection using a semi-autonomous robot fleet. The core of this project is the creation of a VR interface that allows operators to immerse themselves in a digital twin of the remote environment, facilitating intuitive and efficient control over robot inspection. This interface supports both third-person and robot's point-of-view perspectives, enhancing situational awareness and decision-making capabilities in hazardous environments.

The software framework is built upon ROS 2 Foxy, and the VR application was designed with a new graphics engine called Wonderland Engine, particularly suited for lightweight WebXR experiences capable of running on a number of VR headsets, like the Oculus Quest 2. The communication between the interface and the robot fleet is tackled by a custom-built WebSocket server.

The work is demonstrated using simulated robot scenarios in Gazebo. The demonstration serves as a proof of concept, showcasing the viability of the VR interface in a controlled environment and setting the stage for future real-world applications.

This work contributes to the field of VR-enhanced remote inspection by providing an interface that bridges the gap between operators and remote environments. The integration of VR technology with robotic systems opens new possibilities for remote operation, offering a more immersive and intuitive control mechanism that can be adapted to various industrial and research applications.

Keywords:

VR, robotics, fleet, interface, HRI, teleoperation, autonomous, inspection, SAR, search and rescue, disaster, AR, XR, WebXR

CERCS: P170 Computer science, numerical analysis, systems, control; T125 Automation, robotics, control engineering

VR-võimendatud kaugkontrolli raamistik poolautonoomsete robotite laevastiku jaoks

Lühikokkuvõte:

Käesolev magistritöö kätkeb virtuaalreaalsuse (VR) põhise kasutajaliidese ja kommunikatsiooni infrastruktuuri väljatöötamist, mille abil saab juhtida poolautonoomset robotparve ohtlikke keskkondade monitoorimise eesmärgil. Väljatöötatud kasutajaliides võimaldab operaatoril siseneda monitooritava keskkonna digitaalsesse kaksikusse, hõlbustades intuiitiivset ja tõhusat kontrolli robotparve üle. Antud kasutajaliides toetab nii kolmanda isiku kui ka roboti vaatenurka, suurendades operaatori olukorrateadlikkust.

Töö tulemusel valminud tarkvararaamistik põhineb robotika tarkvaraarenduse platvormil ROS 2. VR kasutajaliidese loomisel kasutati graafikamootorit Wonderland Engine, mis liidestati Oculus Quest 2 VR peakomplektiga läbi veebilehitseja põhiste VR rakenduste liidestamise teegi WebXR. Graafikamootori ja robotite vaheline suhtlus teostati töö käigus loodud WebSocket serveri abil.

Tööd demonstreeritakse simuleeritud keskkonnas, kasutades robotikasimulaatorit Gazebo. Demonstratsioon näidatab väljatöötatud VR-liidese rakendatavust kontrollitud keskkonnas, mis loob eeldused tulevaste reaalsete rakenduste jaoks.

Võtmesõnad:

VR, robotika, robotipark, kasutajaliides, teleoperatsioon, autonoomne, kontroll, otsingu- ja päästetööd, katastroof, AR, XR, WebXR

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine; T125 Automatiseerimine, robotika, juhtimistehnika

Contents

1	Introduction	9
1.1	Problem Statement	10
1.2	Objectives	11
1.3	Structure	11
2	Background and Related Work	12
2.1	State of the Art Robotics in Disaster Management	12
2.1.1	Search and Rescue	12
2.1.2	Environment Monitoring	14
2.1.3	Inspection	14
2.1.4	Human-Machine Interfaces	14
2.2	VR Teleoperation	18
2.2.1	Space Robots	18
2.2.2	Underwater Robots	20
2.2.3	Construction Robots	20
2.2.4	Disaster Management Robots	21
2.3	VR outside the gaming domain	22
3	Requirements	24
4	Architecture	25
4.1	VR graphics API	28
4.1.1	WebXR	28
4.1.2	Wonderland Engine	29
4.2	Robot Setup	33
4.2.1	ROS 2	33
4.2.2	Robot Compatibility	33
4.3	Communication Infrastructure	35
5	Interface Design	38
5.1	Menu	40
5.2	Sensor Screen	48
5.3	Robot Component	48
5.4	Teleoperation, Goal Pose, Teleportation and Snap Turn	49
5.5	Screen & Robot Camera Grabber Components	50
5.6	Switching from VR to robot POV	52

6	Demonstration	56
6.1	Hardware setup	56
6.1.1	Oculus Quest 2	56
6.1.2	Lenovo Ideapad 5 Notebook	56
6.2	Software setup	57
6.2.1	Gazebo simulation	57
6.3	Demonstration in Simulated Environment	60
7	Discussion	64
7.1	Hand Tracking	64
7.2	Correcting SLAM generated maps	66
7.3	Why not Unity?	66
7.4	Live Video Stream	68
7.5	Future Work	68
8	Conclusion	70
	Acknowledgments	71
	References	72
	Appendix	75
	Source Code	75
	Running the Interface	75
	GitHub Repository	76
	Licence	77

List of Figures

1	ANYmal robot deployed in challenging hazardous environments, from [6], cited by [1].	13
2	Amphibious robot Krock2, which uses a sprawling posture for crawling locomotion on land, is able to swim in water, and can maneuver through tight spaces using coordinated limb and spine actuation, from [7], cited by [1].	13
3	a) NIFTi UGV in Mirandola, Emilia- Romagna region, Northern Italy after the 2012 earthquakes [5]; b) Pipe-overheating inspection by an UGV [2]; c) The Telex mobile manipulator robot with the on-board sampling probes in the back [8]; d) Inspection of infrastructure with multirotor UAVs (Ascending Technologies, 2018), cited by [1].	15

4	NIFTi interface [5].	16
5	Telemax Operator Control Unit [8].	16
6	Human–robot interface in which the operator uses pointing gestures, estimated from sensors worn in armbands, to provide navigation commands to both flying and legged robots, from [9], cited by [1].	17
7	The real manipulator gripping a solar panel on the left, and the predictive visualization on the right [10].	19
8	Creating an environment model from a robotic 2D image survey by registering to known objects (satellite) and reconstructing unknown or imprecisely known objects (multi-layer insulation hat) [11].	19
9	View of the gripper from TWINBOT, the black box model changes color after the operator puts the gripper in the right position [13].	20
10	The simulated Brokk demolition robot and an operator teleoperating it from the VR setup [14].	21
11	High-level overview of a novel immersive robot teleoperation and scene exploration system where an operator controls a robot using a live captured and reconstructed 3D model of the environment [15].	23
12	CERN mixed reality human-robot interface - operator’s view from an LHC intervention scenario [16].	23
13	Hardware overview of the setup	25
14	Software architecture of the interface	27
15	Lifetime of a web VR app.	29
16	Wonderland Engine’s object hierarchy tree	31
17	Properties tab and component settings of a game object	32
18	List of running nodes for a Turtlebot3 Waffle robot namespaced Waffle1.	34
19	The Turtlebot3 Waffle (Photo: Erico Guizzo/IEEE Spectrum).	34
20	Implemented communication pipeline a) between the robot’s navigation stack and the interface, b) between the interface’s goal pointer and the robot’s navigation stack.	37
21	Sample scene in Wonderland Engine	38
22	The interface’s floating menu a) in the Free Roam mode and b) in the Robot POV mode.	40
23	VRChat’s floating menu.	41
24	Comparison between the Button and Switch components.	42
25	Relation between Switch and Switch Handler with the Emitter component.	43
26	Events following a Robot POV mode button press.	44
27	Events following a Free Roam mode button press.	45
28	Events following a Robot1 switch press.	47
29	The sensor screen showing information for Robot 1 when it is selected on the menu.	48

30	Snapshot command process.	50
31	Camera frame request process.	51
32	Preparing the hemisphere screen for wide-angle images with Blender's UV-Map editor.	53
33	Wonderland Engine's editor with the hemispheric screen and a camera frame applied as texture.	54
34	Summary of the conversion of camera frames to textures.	55
35	The Quest 2 and its Touch controllers.	57
36	Turtlebot3 Waffle robots performing SLAM (Gazebo on the left, Rviz on the right)	58
37	A wide-angle image from the TB3 Waffle's simulated fish-eye camera. .	59
38	Blender preview of the demonstration scene.	61
39	Close-up of the demonstration's three tasks: a) inspection of a chemical storage room, b) inspection of a power distribution room, c) inspection of a spoiled water exhaust pipe.	61
40	Operator's point of view in VR for the three tasks: a) inspection of a chemical storage room, b) inspection of a power distribution room, c) inspection of a spoiled water exhaust pipe.	62
41	Robot's point of view for the three tasks: a) inspection of a chemical storage room, b) inspection of a power distribution room, c) inspection of a spoiled water exhaust pipe.	63
42	a) Teleportation based on pointing gestures. b) Wrist button concept to open the menu with hand tracking.	65
43	Meta's Horizon Worlds wrist "wearable" user menu.	65
44	Comparison between a raw SLAM map and its edited version	66
45	Sample scene in Unity Engine, editor view on top and VR render at the bottom	67

Nomenclature

2D	Two Dimensional
3D	Three Dimensional
AMCL	Adaptative Monte Carlo Localization
API	Application Programming Interface
AR	Augmented Reality
CPU	Central Processing Unit
DOF	Degrees Of Freedom
FPS	Frames Per Second
GPU	Graphics Processing Unit
HRI	Human-Robot Interaction
OS	Operating System
PC	Personal Computer
POV	Point Of View
ROS	Robot Operating System
SLAM	Simultaneous Localization And Mapping
SVG	Scalable Vector Graphics
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
URDF	Unified Robotics Description Format
USB	Universal Serial Bus
VR	Virtual Reality
WLE	Wonderland Engine
XR	Extended Reality
YAML	Yet Another Markup Language

1 Introduction

Robotic technologies are commonly deployed in the realm of search and rescue (SAR), more often in the aftermath of a catastrophe [1]. Disaster management can be broken down into four steps, one of which is the prevention of future disasters, and the mitigation of their consequences [1]. Deployment of remotely operated robots, equipped with sensors and cameras, is a response to the increasing need for preventive measures [2]. Neglected issues in structures and industrial facilities can lead to significant disasters, often due to overlooked inspections [2]. Certain environments are too dangerous or remote for human assessment, while others necessitate temporary plant shutdowns for human inspection, which disrupts regular operations [2]. Miura et al. [2] suggest that the utilization rate of unmanned inspection solutions should be increased since they allow examinations to be carried out with plant facilities still in operation. However, Delmerico et al. [1] argue that if such robotic systems transcend human perception with their sensors, and exhibit situational awareness beyond what the operators naturally obtain, experts will prefer semi-autonomous technologies, where the robot is used as a tool supposed to augment the human-factor rather than eliminating it totally. Full autonomy is said difficult due to poorer adaptability to complex environment, and on the other hand, full manual control is too much of a cognitive load for the operators [1]. Opting for either autonomy or manual teleoperation depending on the situation appears to be a balanced approach [1].

It is understood that collaborative robot teams - teams where robots and human operators interact closely - represent the optimal solution for Disaster Management. However, the success of such teams hinges on the existence of a robust and intuitive human-machine interface. Experts and professionals of the field agree that the key criteria for human-machine interfaces are the ease of use, simplicity and affordability of the systems [1]. The operators should receive minimal training and their cognitive load should be reduced to the minimum [1]. Off-the-shelf solutions are preferred over specialized equipment as they offer a greater reliability, and are simpler to use [1]. Also, the use of natural interactions like pointing gestures could contribute to the accessibility of human-machine interactions [1]. That being said, a scenario where the operator can perform proximity interaction in direct line-of-sight with the robot (ex: pointing at a location for the robot to explore) is hardly imaginable in the aforementioned difficult or life threatening conditions.

1.1 Problem Statement

The characteristics of the environment can render the tasks of multimodal robot teams hazardous, potentially jeopardizing the mission in certain instances [2]. This risk may make close interaction impractical, resulting in a diminished experience for the operator [1].

Virtual Reality is seen as a future trend in the realm of HRI [3]. In that regard, this thesis proposes a human-robot interface using Virtual Reality which allows operators to navigate a digital twin of the robot fleet's environment, where 3D visuals of each robots can be seen navigating and conducting tasks autonomously, while still having the option to switch the robots' perspective individually by accessing their cameras. To better illustrate the purpose of this work, the following concrete scenario is defined:

In an industrial facility with varied spaces, a fleet of three robots perform individual inspection tasks:

- The first robot is tasked with the inspection of a chemical storage area.
- The second robot inspects the temperature of electrical panels in a power distribution room.
- The third robot inspects the plant's spoiled water exhaust pipe.

In all three cases, the environment is too hazardous for human inspection. Therefore, a remote operator monitors the robot's actions from the VR interface, and reacts based on the robot's warnings. In all cases the operator takes action as follows:

- The first robot detects high levels of gases with its gas sensor in the chemical storage area, the operator requests a visual from the robot's camera and notices a spill on the floor underneath one of the barrels.
- The second robot inspects the electrical panels with a thermal camera. Visualizing the camera footage reveals that none of the panels has reached abnormal temperatures.
- The third robot detects abnormal humidity levels in the vicinity of the pipes with a moisture sensor. The operator requests a visual and notices a leak under the pipe.

1.2 Objectives

Based on the needs outlined in the problem statement, this thesis establishes the following goals:

1. Analyze the VR interface designs suited for remote inspection.
2. Design a VR interface for remote inspection that scales to multi-robot systems.
3. Design the underlying implementation to leverage widely adopted robotics frameworks, such as ROS.
4. Provide a demonstration of the prototype interface with a simulated fleet of robots.

1.3 Structure

This thesis begins by providing a background and reviewing related work to contextualize its contributions within the current state of the art in robot remote inspection and VR teleoperation solutions. Following this, it outlines the project's requirements, drawing from existing inspection solutions. A dedicated section then details the proposed framework's architecture, highlighting key hardware and software components. Another section delves into the VR interface's design process. Challenges encountered, critical decisions made, and practical insights are discussed in a subsequent section which also suggests future directions for enhancing the framework. The thesis concludes by summarizing the project's achievements and results.

2 Background and Related Work

This section's goal is to lay the groundwork for the thesis by reviewing existing robotics applications in disaster management and remote inspection. Additionally, it examines prior research on the current state of VR/AR technologies in robotic teleoperation. Having a clear understanding of the current state of the art is crucial for identifying the gaps in research and the needs that the solution proposed by this thesis aims to fulfill.

2.1 State of the Art Robotics in Disaster Management

The current state of robotics in disaster management is a dynamic and evolving field, marked by advancements in various aspects of robot design, control, and human-robot interaction. The integration of robotics into disaster management operations has the potential to significantly enhance the efficiency and safety of rescue missions.

Schneider et al. [4] define three areas of applications of robotics in disaster management, which are discussed in the next subsections:

1. Search and Rescue
2. Environment monitoring
3. Inspection

2.1.1 Search and Rescue

Search and Rescue missions, in response to disasters, imply navigating in unstructured and unknown environments, often containing obstacles [1] that cannot be negotiated by human first responders [4]. Special ground robots (tracked, legged or wheeled) are deployed to maneuver into those difficult terrains, and perform missions such as searching for victims in collapsed buildings or putting out fires (Figure 1). Aerial robots, UAV, are also deployed since they can navigate to unreachable parts of buildings and fly through narrow gaps with fine precision [1]. UAVs used as flying cameras can be used to provide a visual of a ground robot to a remote operator [1],[5]. Recent research has also focused on bio-inspired robot designs, which offer promising applications in SAR scenarios due to their adaptability to different environments and terrains (Figure 2) [1].



Figure 1. ANYmal robot deployed in challenging hazardous environments, from [6], cited by [1].

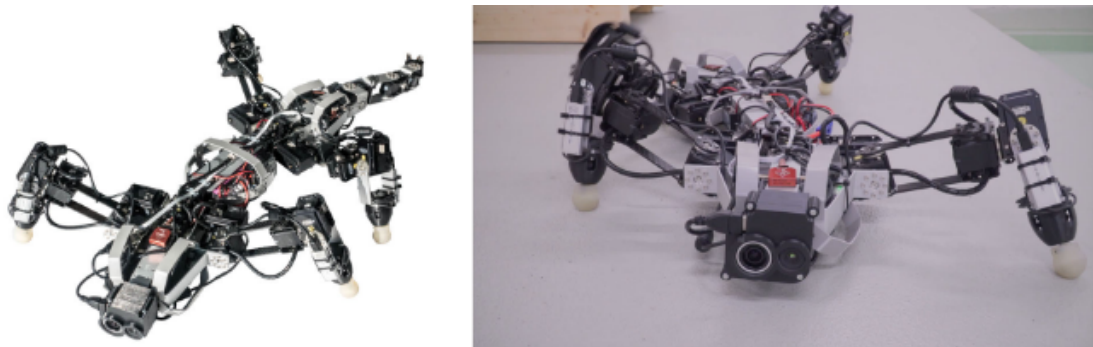


Figure 2. Amphibious robot Krock2, which uses a sprawling posture for crawling locomotion on land, is able to swim in water, and can maneuver through tight spaces using coordinated limb and spine actuation, from [7], cited by [1].

2.1.2 Environment Monitoring

Environment Monitoring relates to the measurement and sensing of parameters within an environment in the aftermath of a disaster, a task that is often too dangerous to be performed by humans [4]. The parameters can be chemical, biological or radiological hazards after incidents [4]. With the example of radiations, which are usually harmful even to robots, operators can build radiation maps by blending images and data collected from radiation detection modules [8]. The resulting maps are used in robot navigation and the updated costmaps, based on radiation levels, allow safe exploration [8]. Robots like the Telemax mobile manipulator robot (Figure 3c), can sample radioactive and dust particles from the environment with probes switched by the operator [8].

2.1.3 Inspection

Inspection is similar to Environment Monitoring in the sense that it consists in assessing precise parameters of an environment, but it is usually a pre-disaster application [4]. If it can be applied to determine the long-term stability of partially wrecked buildings [4], as demonstrated by the deployment of UGVs and UAVs in Mirandola by the NIFTi consortium [5] (Figure 3a), inspection robots are more commonly deployed to inspect building integrity, radiation levels and chimneys in the nuclear industry, in prevention of incidents [4]. UAVs are appreciated for inspection of large infrastructure from the air (Figure 3d), as they can provide detailed 2D or 3D maps of the environments [1]. But UAVs' effectiveness falls short when the inspection task is carried out in confined areas with obstacles such as piping, where UGVs are preferred [2]. Another benefit of using ground robots is that they can carry manipulators, along with cameras, and interact with the environment: Miura et al. [2] made a robot capable of reading gauges, observing rust on pipelines, and the manipulator was designed to operate valves (Figure 3b).

2.1.4 Human-Machine Interfaces

The substantial research in robotic technologies led to the deployment of a myriad of effective solutions that are in use for disaster management operations, as demonstrated by the previous sections. Whether it is for monitoring, recovery or inspection, rescue workers and robot operators have means to accomplish their missions, but the effectiveness of their work depends on the interactions with the robot agents.

As shown by Figures 4 and 5, most setups rely on traditional graphical user interfaces, 2D screens, with joysticks and buttons as controls. However, [1] presents alternative approaches to teleoperation and HRI with technologies like the smart glove, used to control UAVs and UGVs, offering a hands-free gesture-based interface relying on sensorized armbands. The operators can point somewhere and the robot goes to the pointed location (Figure 6). The objective of such novel interfaces is to relieve the operator's cognitive

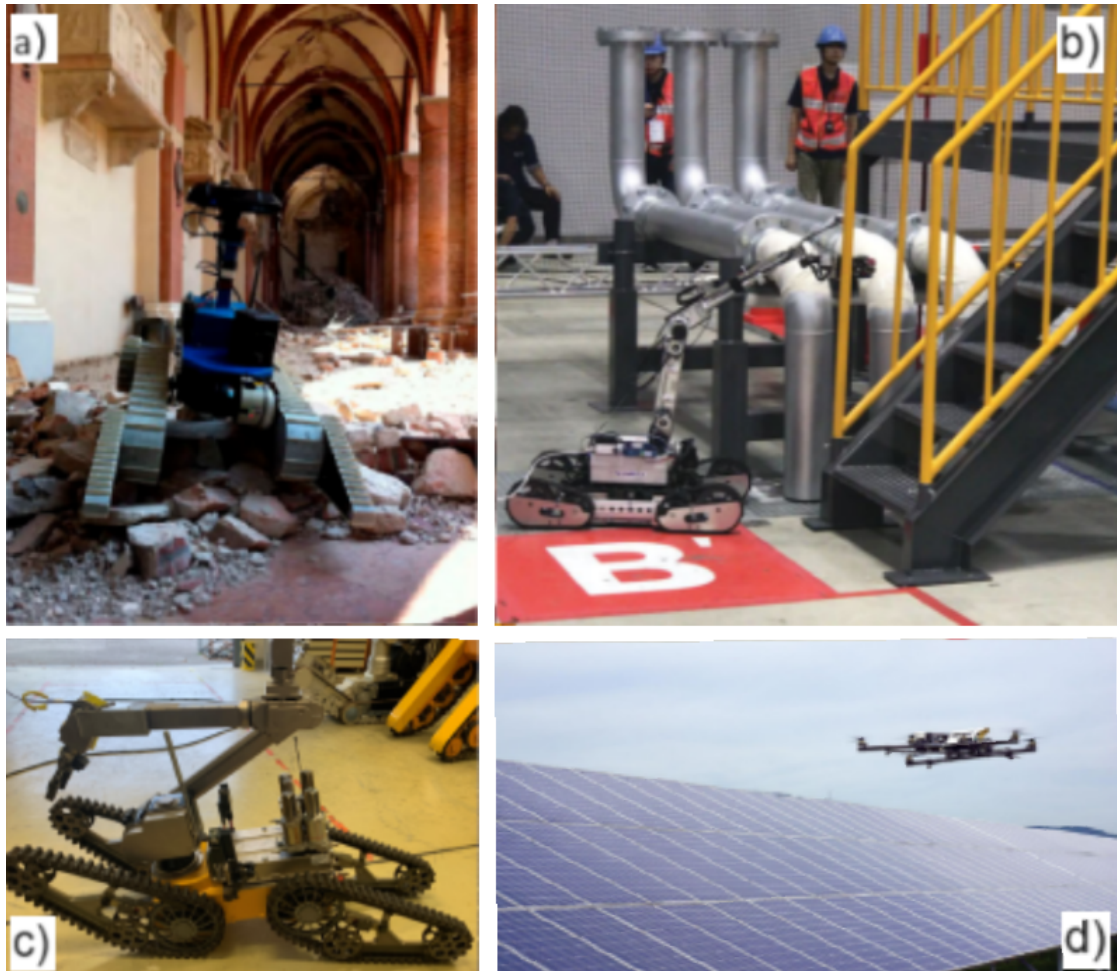


Figure 3. **a)** NIFTi UGV in Mirandola, Emilia- Romagna region, Northern Italy after the 2012 earthquakes [5]; **b)** Pipe-overheating inspection by an UGV [2]; **c)** The Telemax mobile manipulator robot with the on-board sampling probes in the back [8]; **d)** Inspection of infrastructure with multirotor UAVs (Ascending Technologies, 2018), cited by [1].

load, SAR operations being already stressful without the addition of a complex system to interact with [1]. Delmerico et al. [1] as well as Schneider et al. [4] agree that the future of SAR robotics lies into the cooperation of human operators and semi-autonomous robots, by the means of engaging and immersive - almost second nature - interactions.

To go further into providing operators with enhanced situational awareness, immersive telepresence through VR is considered, as the illusion of ‘being in’ the robot could well provide the operator with much more natural sense of the robot’s position and its immediate surroundings [4].

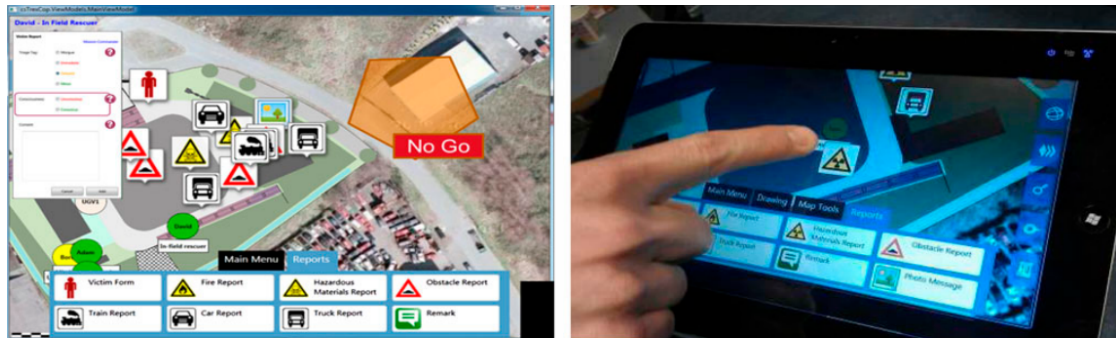


Figure 4. NIFTi interface [5].



Figure 5. Telex Operator Control Unit [8].



Figure 6. Human–robot interface in which the operator uses pointing gestures, estimated from sensors worn in armbands, to provide navigation commands to both flying and legged robots, from [9], cited by [1].

2.2 VR Teleoperation

Recent research in the field of human-robot interaction has given rise to a number of new interfaces based on VR and AR, for teleoperation and collaboration purposes. The following subsections showcase some of those interfaces:

2.2.1 Space Robots

Teleoperating space robots from Earth is difficult due to the significant time delays involved. To solve this, Zainan et al. [10] introduced a VR/AR "predictive display" (Figure 7) that lets operators control a 3D virtual model of the robot. This model is superimposed on real-time camera footage from the robot's location, enabling operators to plan movements before they are actually performed by the robot. The system uses a 4-DOF robotic arm that follows the movements of its virtual counterpart.

Kazanzides et al. [11] also presented a similar predictive display (Figure 8) but used Augmented Virtuality (AV). In this setup, operators work within a completely virtual environment that includes 3D models of the robot and its surroundings, but with the addition of live video feeds. This approach also helps in overcoming the delay issues.

Both of these systems [10] and [11] effectively deal with the time delay problem. However, aligning the robot's real-world movements with its virtual representation remains a complex task. For example, in [11], the robot is tasked with precisely cutting a satellite's thermal insulation layer. Due to the high level of accuracy needed, the operation is semi-automated: the operator issues high-level commands, but the robot carries out the precise cutting, relying on force feedback and various sensors for accuracy.

In [10], the experiments concluded with discrepancies between the model and the real robot's movements. In [11] the results led to the conclusion that AV significantly improved the operator's performance by enhancing their situational awareness, and allowed them to "precisely specify intended motion". However, the operators seemed to prefer sacrificing a better visualization - offered by the VR setup - for their conventional interfaces which they had trained on for years. The VR interface was based on the da Vinci Research kit, which was also used for remote surgery [12], and although the system offered a 6-DOF motion control, it was not the best fit for some of the tasks the operators had to perform. The authors conclude that the operators should be given the choice between several control interfaces to best carry out the interventions.

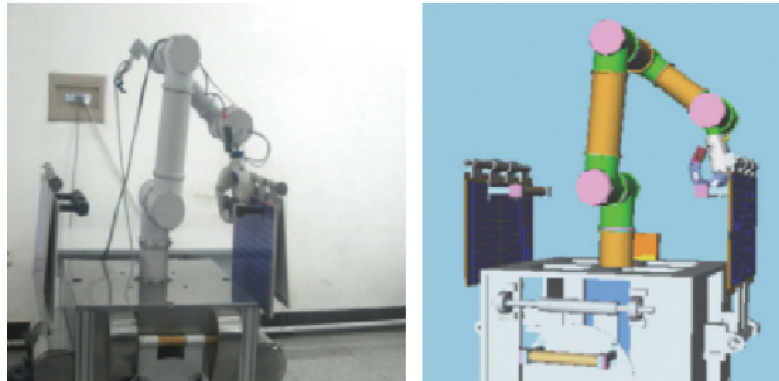


Figure 7. The real manipulator gripping a solar panel on the left, and the predictive visualization on the right [10].

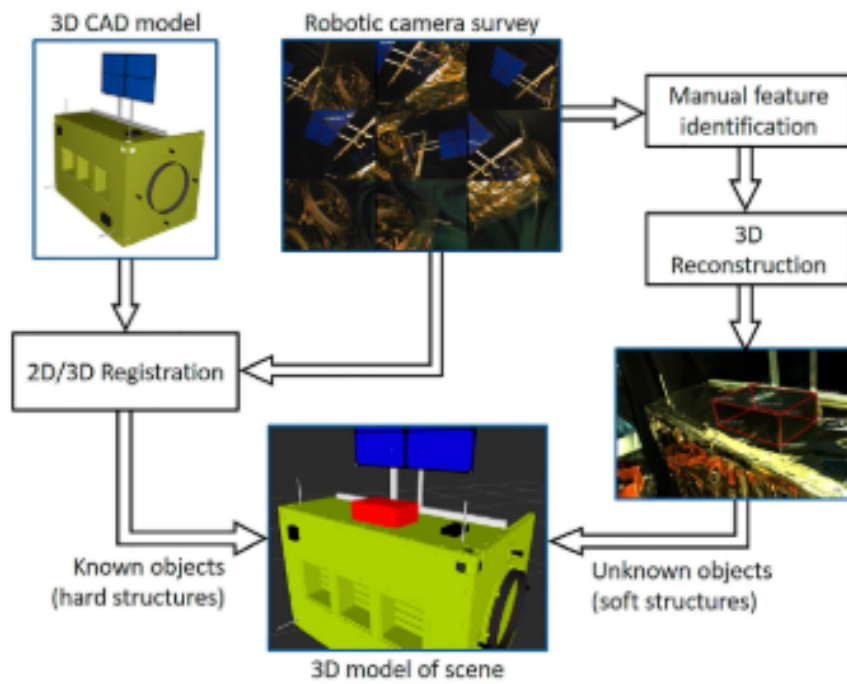


Figure 8. Creating an environment model from a robotic 2D image survey by registering to known objects (satellite) and reconstructing unknown or imprecisely known objects (multi-layer insulation hat) [11].

2.2.2 Underwater Robots

In the context of underwater interventions, the need for which is constantly increasing, remote operated vehicles (ROV) controlled from support vessels are commonly deployed [13]. De la Cruz et al. [13] propose a VR-based human-robot interface which allows to control the robots, and provides visual feedback.

The control system is split into two groups, one for managing the vehicle and the other for managing the manipulator. The operator can switch from either a view of the vehicle or the robot arm. The particularity of this project is that it focuses on the robotic system called TWINBOT, composed of two underwater robots designed for underwater maintenance (oil wells, pipes, observatories etc.). The VR interface is proposed to eliminate the need of a duplicate control setups to control the TWINBOT, it is designed to allow the operator to control both robots at the same time.

The interface was developed and tested on consumer-grade hardware and software like the HTC Vive VR headset, the Unity game engine and several different models of Graphics Processing Units (GPU).

Iterating their work based on usability tests on ROV pilots in simulated missions, the authors have managed to produce a subjectively immersive interface with specific features like a color indicator confirming the correct positioning of the gripper on a target (Figure 9), supposedly helping the operator, or information about the ROV's depth etc. However, lack of realism in the simulation (uncertainty was missing in the simulated environment) and limited performances were reported to be detrimental to complete immersion.



Figure 9. View of the gripper from TWINBOT, the black box model changes color after the operator puts the gripper in the right position [13].

2.2.3 Construction Robots

The use of robotics is emerging in the construction industry, but this creates an urgent need for the construction workers to reskill and learn how to operate new construction robotic solutions [14]. Adami et al. [14] investigate the effectiveness of a VR-training

platform. With a VR setup composed of an HTC Vive and a VR treadmill, and a virtual construction site modeled inside the Unity engine, some construction workers were trained on a virtual replica of a Brokk demolition robot (Figure 10). Other participants of the study received an in-person training instead. The study's results revealed that the workers trained in VR displayed a greater knowledge of the technical and safety procedures related to the robot's operation than those who attended the in-person training. The authors explain that the effectiveness of the VR training resides in the fact that trainees are less hesitant to perform maneuvers since their mistakes in VR have no real consequences. The ability to train with a demolition robot without being exposed to the inherent dangers is the force of such a training tool [14].



Figure 10. The simulated Brokk demolition robot and an operator teleoperating it from the VR setup [14].

2.2.4 Disaster Management Robots

Stotko et al. [15] address the lack of immersion from standard video-based approaches, by proposing a novel VR-interface based on RGB-D (RGB Depth) captured data (Figure 11). The data allows the reconstruction of the robot's surroundings in the VR headset into 3D models. The purpose of this interface is to allow the operator to explore a hazardous, contaminated or inaccessible place like a disaster or industrial site. The strength of this system lies in the fact that the operator can explore the reconstructed environment in VR, independently of the robot's current position. The robot can be teleoperated to scan the area and reconstruct it, the operator can visualize the scene from the robot's perspective, but the operator can arbitrarily explore another part of the reconstructed environment from the third person view, and visualize a model of the robot based on the estimated camera pose.

This system relies on a Kinect RGB-D camera, mounted on a 6-DOF robotic arm,

which itself is mounted on a mobile platform. The processing of the RGB-D data and conversion into the reconstructed 3D environment is tackled by high-end GPUs.

The system was tested by 20 participants and the article concludes by reporting a clear improvement in terms of situational awareness and teleoperation with more precise maneuvers and obstacle avoidance.

Similarly, Szczurek et al. [16] propose a Mixed Reality teleoperation interface (Figure 12) which is aimed at reducing the operator's cognitive load while performing inspection tasks in hazardous environments. This project was conducted at the European Council for Nuclear Research (CERN), and was driven by the need for robotic inspections in environments like the Large Hadron Collider. The interface provides immersive visual feedback with a reconstitution of the robot's surroundings based on point cloud data. The operator can operate the mobile platform's movements or the manipulator that it carries. The operator can also define trajectories that the robot follows autonomously, in an attempt at reducing the cognitive load associated with the task.

The project approached critical challenges associated with interventions in hazardous environments such as positional precision, collision avoidance and network bandwidth and delays. The interface also lets the operator toggle between different camera acquisition techniques (point cloud, 2D camera) and change the streaming settings, to tailor the immersion and deal with the network's congestion due to the complexity of point cloud data transfer.

The paper concludes that the proposed interface does improve the operator's situational awareness, and the ability of changing visualization parameters on the fly helps with optimizing network load.

2.3 VR outside the gaming domain

If gaming paved the way for VR headsets' democratization, VR technologies' are increasingly finding applications beyond entertainment, particularly in the workplace [17]. The VR market is expected to grow significantly, and sectors like healthcare, education, and retail are already exploring its potential [17]. VR and AR technologies can enhance the sense of presence and collaboration in the workplace, suggesting a shift towards more immersive and productive work environments facilitated by VR [17]. The future of VR in work is also suggested by the introduction of Apple's Vision pro, the giant's take at Mixed Reality, a technology bridging AR and VR.

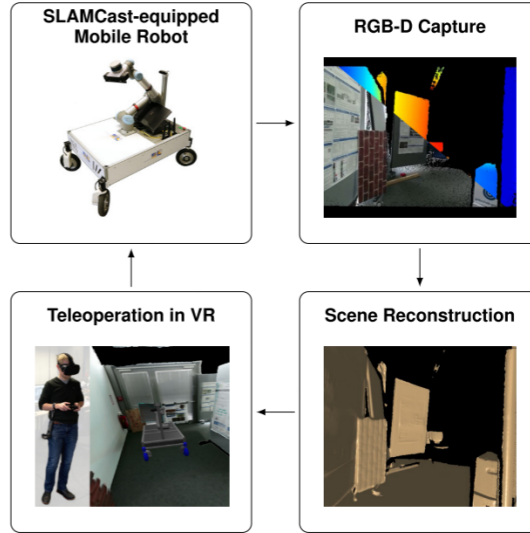


Figure 11. High-level overview of a novel immersive robot teleoperation and scene exploration system where an operator controls a robot using a live captured and reconstructed 3D model of the environment [15].

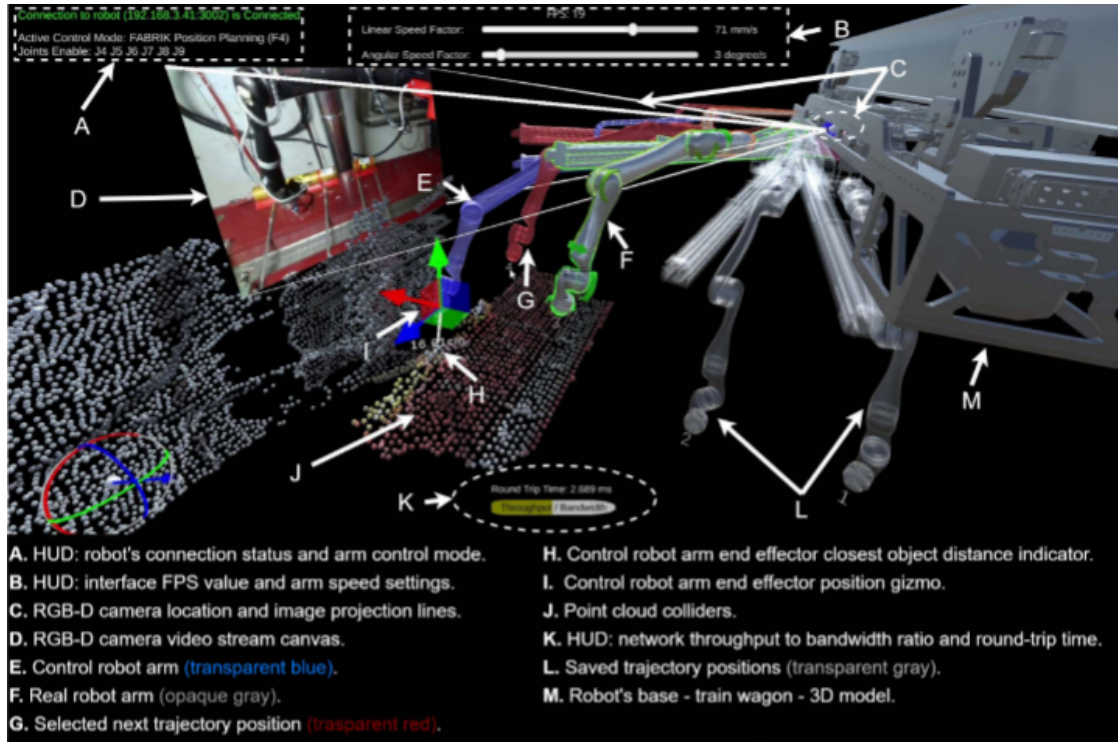


Figure 12. CERN mixed reality human-robot interface - operator's view from an LHC intervention scenario [16].

3 Requirements

The interface proposed by this thesis should satisfy several requirements, in line with the identified needs and gaps in current robotic solutions, applied to inspection in the context of disaster prevention:

1. The interface should allow for semi-autonomous robotic operation and direct teleoperation.
2. The setup should be composed of accessible, off-the-shelf components.
3. The interface should be compatible with most VR headsets.
4. The bridge between the robot's software and the interface should be robust, yet straightforward to maintain and scale up.
5. The software used for the development of the interface should support deployment of prototype iterations.
6. The interface should enhance the operator's situational awareness, combining information from the virtual and the real world. And it should involve natural gestures to maximize accessibility and minimize operator training.

4 Architecture

This section first gives a high-level overview of the architecture, then it delves into the subtleties of its core elements.

As illustrated by Figure 13, the framework relies on three core elements:

1. The operator, equipped with VR gear in a safe environment.
2. The server, host of the VR interface's data and WebSocket server.
3. The robot fleet, in the remote environment.

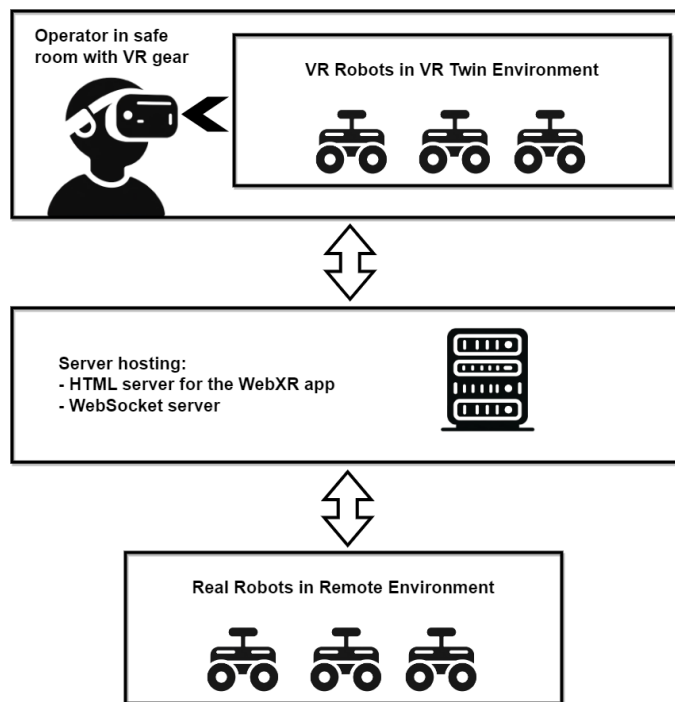


Figure 13. Hardware overview of the setup

The operator, equipped with the VR gear is projected in a VR replicate - a "digital twin" - of the remote environment, rendered within the headset following a request to the server for the application data (3D models, textures, logic etc). The operator is able to visualize the robots upon reception of the real robots' coordinates from the server. The operator can also arbitrarily switch between semi-autonomous control, by setting goal positions with VR controllers or hand movements, or full control by directly teleoperating the robots' movements with the VR controllers. The virtual environment can also be enhanced by requesting camera snapshots of the live robots' surroundings.

The server acts as a communication medium between the VR interface and the actual robot fleet in the remote environment. At the center of the interaction, it hosts an HTML server providing the VR headset with the necessary data for the operator to enter the VR world and interact with it (wired or wirelessly), but it also hosts a WebSocket server which relays robot information (localization, velocities, sensor data etc.).

The real robots in the remote environment perform autonomous navigation based on a map in which they are localized. They can be programmed to perform routine tasks, which can be interrupted by the operator to follow new plans, or even resign of their autonomy and let the operator control them with precision. Each robot establishes connection with the WebSocket server.

Figure 14 summarizes the software architecture of the project. The VR interface graphics are rendered within the headset's native web browser for a standalone headset, or the computer's web browser for a PCVR headset. The server provides the VR headset with graphics via the HTML server hosting a WebXR application. This application also reads controller input that interact with the interface's elements. From the interface, the operator sends high level commands to the robots (goal positions and motor velocities) via the WebSocket server. The latter, implemented with Node.js, speaks to Node.js clients for ROS 2 via the RCLNodeJS library. This library subscribes and publishes to the ROS 2 topics involved in the teleoperation and autonomous navigation of the robot fleet. The communication through the WebSocket server is bidirectional: RCLNodeJS clients also send the robot's transforms and camera frames to the interface.

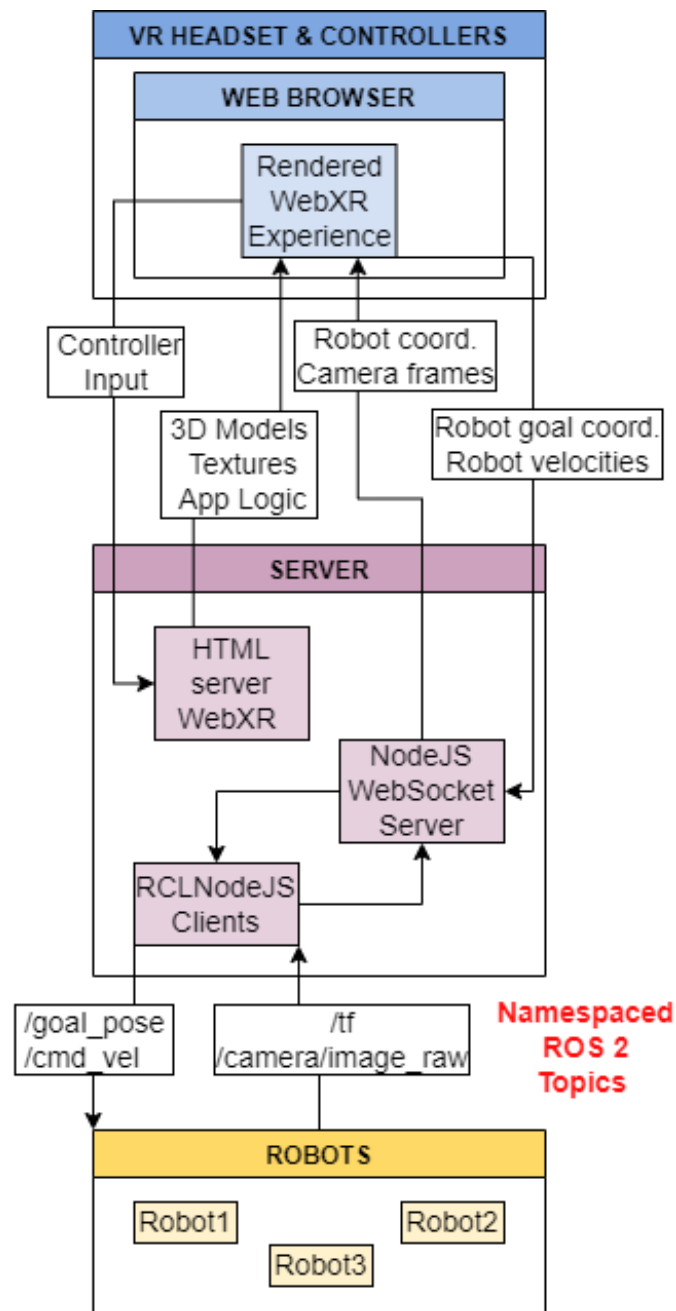


Figure 14. Software architecture of the interface

4.1 VR graphics API

The aim of this section is to introduce the WebXR application programming interface (API) that is used to render the VR content inside the operator's headset. The section also gives an insight on the Wonderland Engine (WLE) graphics engine which is tailored for WebXR.

4.1.1 WebXR

This subsection explains the rationale behind choosing WebXR for developing the VR-enhanced user interface and how it aligns with the project's objectives of creating a universally accessible and scalable VR platform.

WebXR at a glance WebXR is an API that allows to develop and host VR/AR experiences that run on a web browser. The API is capable of detecting the kind of hardware that is trying to access the content via browser requests, and deliver compatible VR or AR experiences accordingly [18] (Figure 15 breaks down the lifetime of a VR web app). It can also register user input from controllers - like the Quest 2 touch controllers - or hand tracking to interact with the 3D objects displayed in the immersive experience [18]. If some VR platforms like the Oculus Quest 2 are marketed primarily as gaming devices, offering numerous games on proprietary app stores, WebXR is not merely designed for game development [18]. As a matter of fact, the API opens up to other uses of AR and VR such as art, data visualization and video [18].

WebXR's advantages WebXR is compatible with phones, and VR/AR devices, whether they are desktop or standalone units. WebXR requires a chromium web browser to work, which is embedded on most modern smartphones, desktop computers, and VR headsets, such as the Oculus Quest 2 [19]. It is a future-proof approach to XR development, VR hardware evolves rapidly as the technology gains traction with the public, but the experiences should remain compatible since they run on the web browser [19]. The same version of an experience can be aimed at both VR and AR devices with minimal code changes to handle the cross-compatibility [19]. Using WebXR, the app build is directly compatible with any VR device as long as the device's web browser is supported (a long list of browsers including Google Chrome is available on WebXR's documentation), but with other engines and VR development methods, the builds are made for specific VR platforms (Oculus, Valve Index, HTC Vive etc.) which means several builds are required at each app update. The said builds need to be installed on the target devices for standalone headsets (the headset holds the computational power), or the computer for PCVR setups (The computer does the graphical computation and the headset is a display).

WebXR development WebXR can be used in many different ways, following the official documentation, experiences can be programmed from the ground up, but the API is also supported by game engines, like Unity [19]. The latter is not natively designed for WebXR, but packages can be imported from a third party package manager [19]. However, WebXR's official page mentions an alternative graphics engine which seemed attractive for this project as it natively supports WebXR out of the box: Wonderland Engine [19].

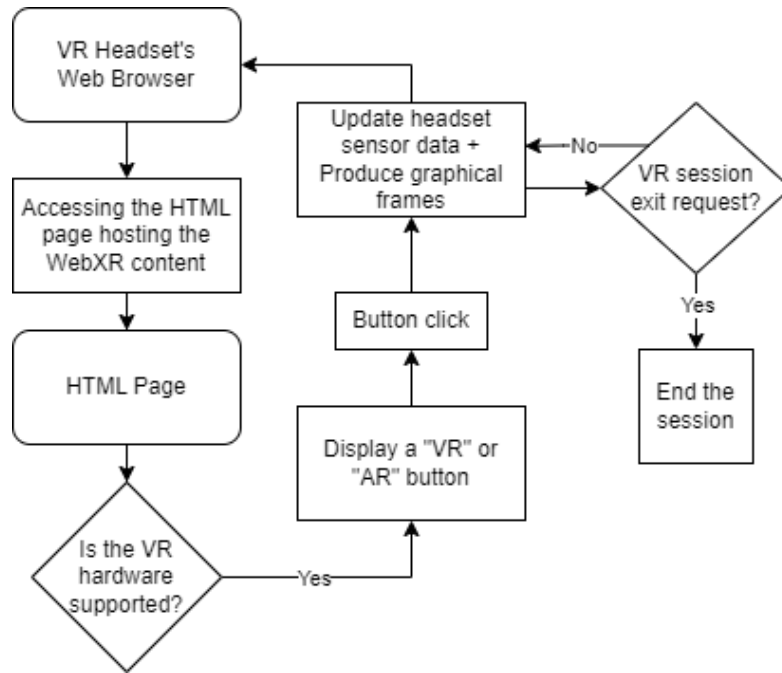


Figure 15. Lifetime of a web VR app.

4.1.2 Wonderland Engine

The Wonderland Engine was selected for its specific optimization for web-based VR experiences and its native support for WebXR. This subsection delves into the advantages of the Wonderland Engine over other VR development platforms.

Wonderland Engine at a glance Wonderland Engine is a highly performant, accessible and lightweight graphics engine for VR and AR on the web [20]. The engine makes development accessible and provides an efficient workflow with features such as reloading the browser to reflect file changes. Its use of WebAssembly (An advanced binary instruction format designed for efficient execution and compact representation of code on modern web browsers) and optimizations such as automatic scene batching allow

to render many objects' visuals without compromising performance (The rendered VR experience does not dip below 60 frames per second even with numerous assets in the scene) [19]. Just like most game engines, it comes with a 3D visual editor in which 3D models can be imported. World models generated with Blender for this project were loaded from the WLE project directory. Various mesh objects can be generated from the editor, such as cubes, planes, cones etc. The editor comes loaded with pre-configured materials, which can be customized (colors, reflections etc.) but they are limited and objects sharing the same material will be applied the same colors. When creating a first project, templates can be chosen from, and the VR template contains the VR-rig - the ensemble of game objects required for the VR headset and its controllers to work - out of the box. WLE internally supports any common VR hardware: most VR controllers are mapped by default, tracking the controllers from the headset does not require any extra effort, and hand tracking is even supported natively. WLE relies on JavaScript and Typescript as scripting languages, the API is well documented on the official website, breaking down every feature with examples, and if the documentation fails to explain a specific concept, the community is highly reactive on the official Discord server.

Wonderland Engine's efficient workflow Wonderland Engine finds one of its strengths in simplifying deployment after code and scene changes: saving the script from the code editor after code changes triggers the engine to compile the whole project. The engine takes full advantage of the wireless Android Debug Bridge (ADB) commands, and if the application is being previewed in the VR headset when changes are made, the browser is refreshed to reflect changes. Thanks to this feature of the engine, it was possible to jump back and forth between the code editor and the head mounted display (HMD) for efficient debugging and feature testing.

Wonderland Editor was selected as best fit for the development of the interface's prototype, for its compliance with the project requirement number 5, but the **Discussion** chapter of this thesis contains a comparison between WLE and Unity.

Scripts and Components in Wonderland Engine In WLE, the scripts are denoted as "components" which can be attached to game objects, whether they are 3D meshes or symbolic "empty" objects that do not contain any visual representation, but play a role in the interaction of the user with the rest of the scene. Game objects are hierarchically related as a tree: each object is a branch of that tree which can contain nested sub-objects with components of their own (see Figure 16). Components are JavaScript classes, and can interact with each other by either belonging to the same hierarchy, or by pointing at each other, by setting the class properties in a way that they call other classes' methods and attributes. Those settings are either hard-coded in the scripts, or dynamically adjusted within the editor as shown in Figure 17.

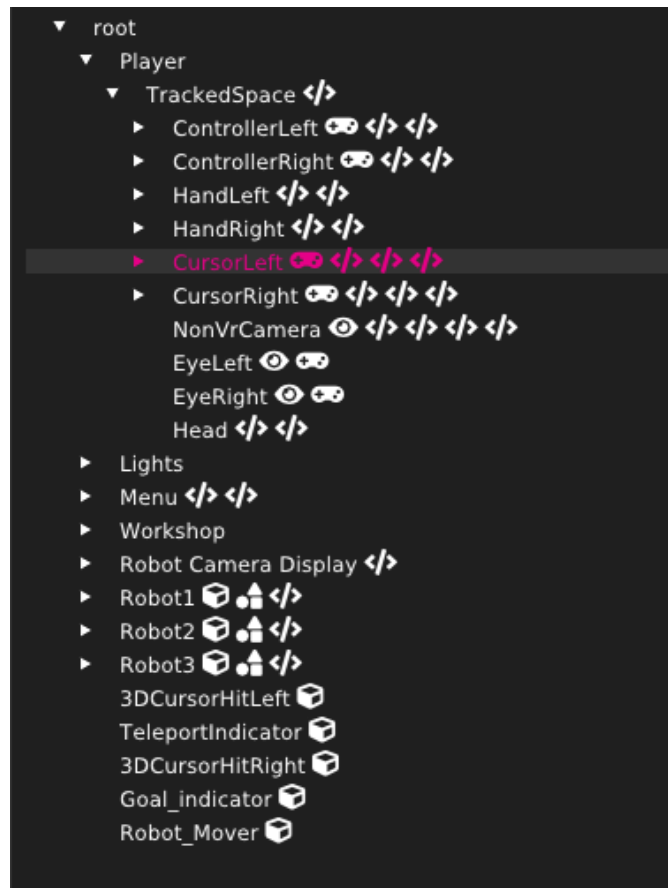


Figure 16. Wonderland Engine's object hierarchy tree

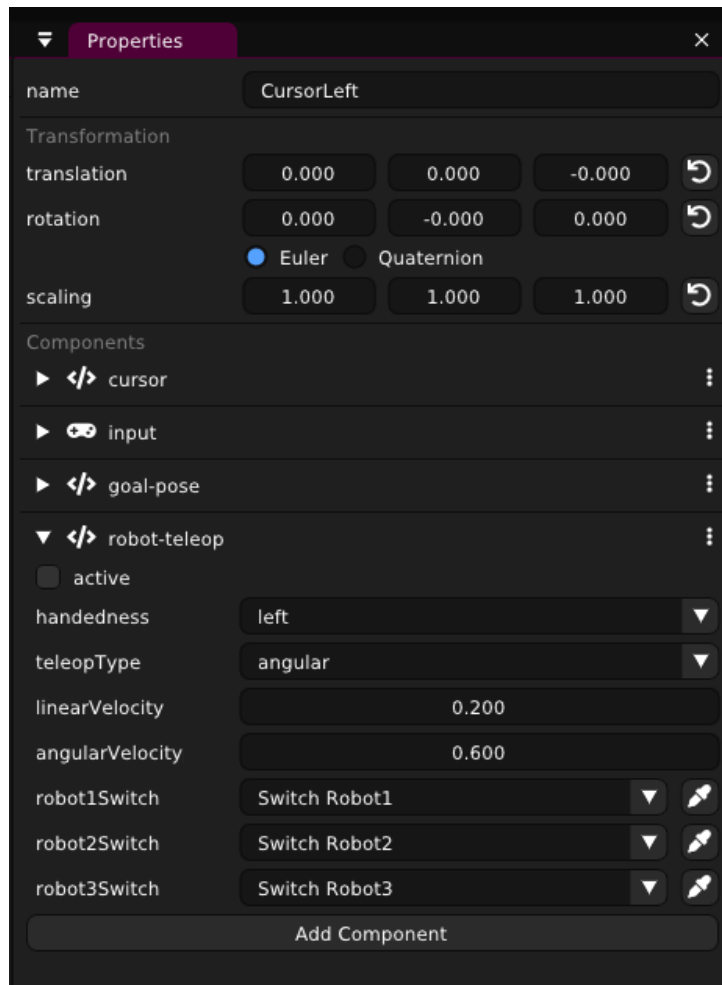


Figure 17. Properties tab and component settings of a game object

4.2 Robot Setup

4.2.1 ROS 2

As stipulated in the requirement number 4 of the project, the bridge between the robots and the interface should be robust and scalable. For this matter the architecture makes use of the ROS middle-ware which is heavily documented and which benefits from official and community maintained packages for most commercially available robots. A plethora of tutorials describe how to implement autonomous navigation for one or more robots. The interface interacts with ROS 2 Foxy. Although ROS 1 is more mature and still very much appreciated by the community as a quick-development research tool, it has reached its end-of-life. Moreover, ROS 2 was introduced as the industrial-grade successor improving on security, reliability and efficiency [21].

4.2.2 Robot Compatibility

This interface was designed to interact with fleets of ROS 2 robots regardless of the robot models, as long as they satisfy the following requirements:

- Each robot must be able to navigate (the interface is not compatible with static manipulators for instance).
- Each robot must support a ROS navigation setup.
- The ROS 2 topics must have namespaces for the messages to be isolated between each robot.
- The robots must publish their transforms and joints to be localized.
- Each robot must publish camera data on a namespaced camera topic.
- Each robot must have a namespaced velocity control topic to be teleoperated.
- Each robot must publish laser scan data from their LiDAR on a namespaced topic.

The Figure 18 shows the list of specific ROS 2 nodes that should be running for each robot to work with the interface.

An example of compatible robot is the Turtlebot3 Waffle (Figure 19). This small mobile robot is commonly used for education and research. It is made for exploration with a built-in 360° LiDAR, a camera and two robust servomotors. It also has a nice payload of 30kgf which means it can carry sensors, a small manipulator etc.

```
gautier@gautier-ubuntu: ~  
gautier@gautier-ubuntu: ~ 80x28  
gautier@gautier-ubuntu:~$ ros2 node list | grep waffle1  
/waffle1/amcl  
/waffle1/amcl_rclcpp_node  
/waffle1/bt_navigator  
/waffle1/bt_navigator_rclcpp_node  
/waffle1/camera_driver  
/waffle1/controller_server  
/waffle1/controller_server_rclcpp_node  
/waffle1/global_costmap/global_costmap  
/waffle1/global_costmap/global_costmap_rclcpp_node  
/waffle1/global_costmap_client  
/waffle1/local_costmap/local_costmap  
/waffle1/local_costmap/local_costmap_rclcpp_node  
/waffle1/local_costmap_client  
/waffle1/planner_server  
/waffle1/planner_server_rclcpp_node  
/waffle1/recoveries_server  
/waffle1/recoveries_server_rclcpp_node  
/waffle1/robot_state_publisher  
/waffle1/transform_listener_impl_5595ea060b40  
/waffle1/transform_listener_impl_55a35ec7a3f0  
/waffle1/transform_listener_impl_55a92c433830  
/waffle1/transform_listener_impl_563e10cc7950  
/waffle1/turtlebot3_diff_drive  
/waffle1/turtlebot3_imu  
/waffle1/turtlebot3_joint_state  
/waffle1/turtlebot3_laserscan  
gautier@gautier-ubuntu:~$
```

Figure 18. List of running nodes for a Turtlebot3 Waffle robot namespaced Waffle1.

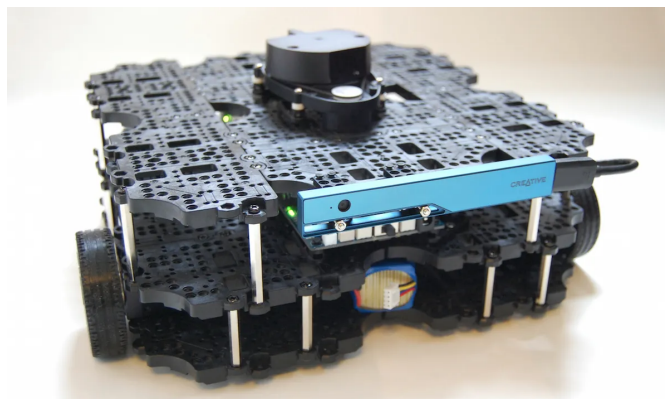


Figure 19. The Turtlebot3 Waffle (Photo: Erico Guizzo/IEEE Spectrum).

4.3 Communication Infrastructure

The VR experience made with WLE and WebXR runs on a web browser, and the application data is hosted by an HTML server which compiles a set of scripts. There is no built-in compatibility between the WLE JavaScript API and ROS 2 which means that a custom-built communication medium had to be implemented. To stay consistent with the rest of the framework, this medium had to be implemented in JavaScript which is why the communication between the robot fleet's middleware, ROS 2, and the interface is facilitated by a Node.js WebSocket server. It is hosted on the same server as the HTML server that supplies the VR application data (as illustrated by Figure 14). The Wonderland Engine VR experience contains the JavaScript clients that interact with the robots through the WebSocket server issuing requests and receiving responses under the form of JSON messages. Having a Node.js server at hands, research on the ROS 2 documentation revealed the existence of the Rclnodejs library.

Rclnodejs is a Node.js client library for ROS 2 [22]. It provides tooling and comprehensive JavaScript and Typescript APIs for developing ROS 2 solutions capable of interoperating with ROS 2 nodes implemented in other languages such as C++ and Python [22]. The library allowed to create ROS 2 subscribers and publishers for the various nodes and topics involved in the autonomous navigation and teleoperation of the robot fleet. The WebSocket server is set up in a way that clients broadcast messages to the rest of the network, except to themselves. The Figure 20 shows two examples of the basic communications handled by the server, one way from ROS 2 to the interface's clients, the other way from the interface to ROS 2.

Two types of messages transit through the WebSocket server: "Stringified" JSON messages, usually conveying coordinates or commands, and Binary large objects (Blobs), used exclusively to convey image data from the robots' cameras.

Since every client in the network is able to broadcast messages and receive them from any peer in the network, the server is experiencing clutter of information which needs to be sorted out. To do so, messages are formatted in a very specific way: all JSON messages have a mandatory key called 'identifier', with its value set with a string relevant to the type of message. For example, the interface runs a client script that requests camera frames. The camera commands are broadcasted as such:

```
{'identifier': 'camera_trigger',  
  'command': 'camera_capture_ON'}
```

Similarly, the goal positions are sent this way to the robots, with the mandatory robot namespace as an identifier prefix:

```
{'identifier':'robot1/goal_pose',
  '0':'x_coordinate',
  '1':'y_coordinate',
  '2':'z_coordinate'}
```

Rclnodejs subscribes to ROS 2 topics such as **/tf** and **/camera/image_raw**, and publishes to topics like **/cmd_vel** and **/goal_pose**. In the case of the *robot-pose-subscriber*, raw transforms from the **/tf** topic are received as serialized ROS 2 messages, then parsed to JSON, and an extra identifier 'robot_pose' is added as key following the same aforementioned principle. Before being sent to the server, the messages as JSON must be converted to strings using the `JSON.stringify()` method which is natively supported by JavaScript. On the receiving end, the messages must be parsed using the `JSON.parse()` method. Then, the identifier is extracted and the processing only executes if the expected identifier is contained in the message. If not, the message is ignored.

If most messages can be conveyed by the WebSocket server as "stringified" JSON, images from the robots' cameras are handled differently. The *robot-camera-subscriber* client subscribes to the ROS 2 **/camera/image_raw** topic and retrieves image messages essentially composed of the image data as an unsigned 8-bit integer array, and information like the encoding (RGB8), the image dimensions etc. The image data array is large for a 1280x720 pixels image, the computation time to convert several hundreds of thousands of values alongside image metadata to string, then parse as JSON is high and introduces a lot of latency between the image request from the interface, the transmission from the server, and the decoding within the interface. Instead, only the data is kept from the raw messages, and this data is sent as a Blob to the server. For that reason, each client expecting messages from the server also verifies the type of message before trying to parse it to JSON. If the received message is of type "object" or an instance of Blob, but the desired message is a string, the message is ignored and the next one is processed. Blobs can also contain identifiers, to separate frames coming from several robots. Strings can be broken down into chars, and each char can be concatenated as buffer to the rest of the image buffer. On the receiving end it is just a matter of slicing the received buffer to isolate the identifier and convert it back to string. But in the context of this version of the interface, only static images of the robots are displayed, from one robot at a time. Meaning only one camera stream is subscribed to at once and there is no need to sort image sources.

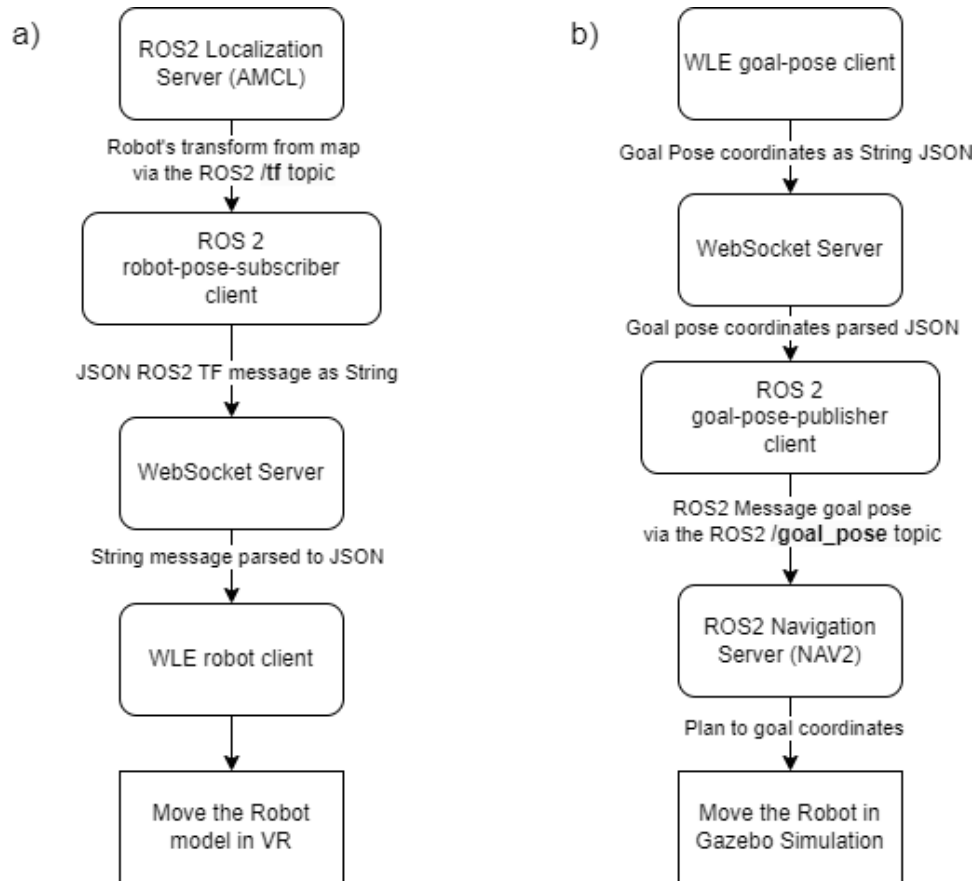


Figure 20. Implemented communication pipeline **a)** between the robot's navigation stack and the interface, **b)** between the interface's goal pointer and the robot's navigation stack.

5 Interface Design

The design of the interface, centered around user experience, underwent two major iterations, each refined through real-user testing. The version presented in this section represents the final design, which was developed by incorporating one user's feedback.

The scene in WLE is based on a 3D model created in Blender. 3D models exported from Blender as **.gltf** format contain materials and textures which are recognized by WLE. Figure 21 displays the scene loaded into the editor.

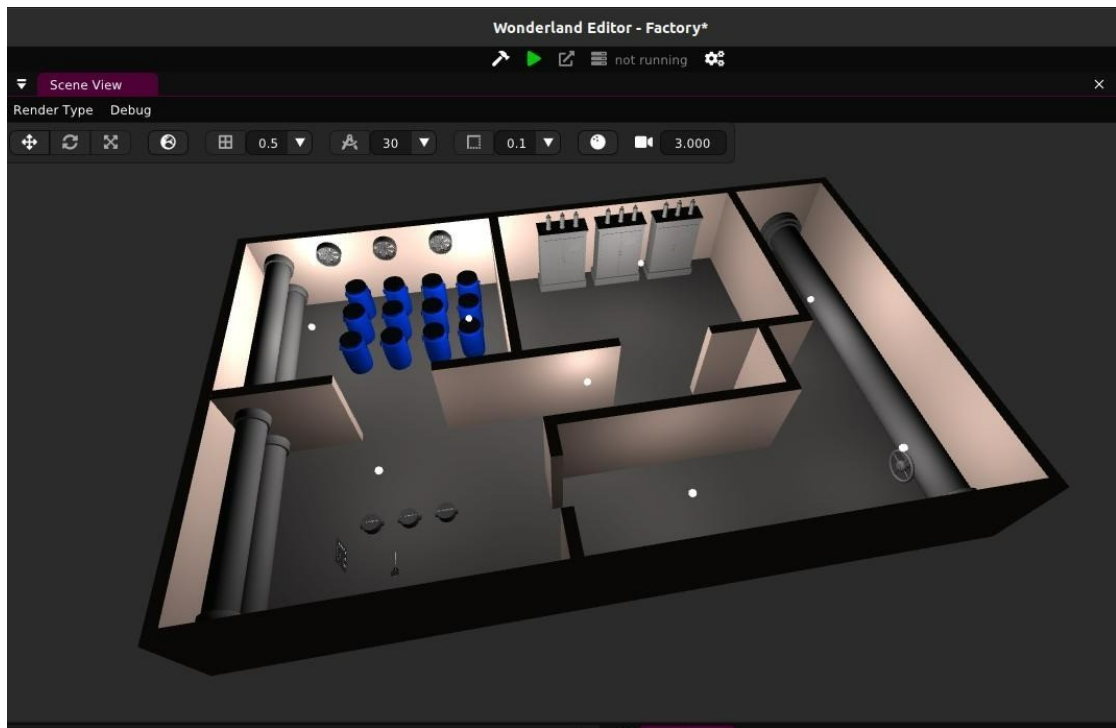


Figure 21. Sample scene in Wonderland Engine

As mentioned in the section 4.1.2 about WLE, the operator interacts with the scene via components which dictate the behavior of the 3D elements that compose the virtual environment. This section focuses on the components that play a significant role in the interface, whether they were shipped by default with the engine, or created specifically for the interface.

Among the default WLE components, the following are used for the interface:

- **Button:** this component triggers events when the operator presses a button mesh on the menu.

- **Teleportation:** this component teleports the user to the location indicated by the cursor.
- **Emitter:** this component allows to send messages between WLE components.

The following components were custom-built for the needs of the proposed interface:

- **Robot:** this component is attached to the robot 3D models, and is responsible for their movements within the scene.
- **Switch:** this component is used on the menu to either select a robot to interact with, or the control mode. This component differs from a button as it behaves as a toggle switch.
- **Button handler:** this component triggers events based on which button was pressed.
- **Switch handler:** this component triggers events based on which switch was toggled.
- **Robot Camera Grabber:** this component sends a trigger to the Screen component with the robot's namespace.
- **Screen:** this component requests a camera frame from the desired robot, and loads it as model texture which is applied to a screen 3D mesh.
- **Menu bring-up:** this component displays a menu facing the user when the associated key is pressed on the VR controllers.
- **Goal pose:** this component sends a goal pose to the robot at the location indicated by the cursor.
- **Teleoperation:** this component allows to manually pilot the selected robot with the VR controller's thumbsticks.
- **Snap Turn:** this component rotates the operator's view at fixed angles within the scene from the push of a thumbstick to the sides.
- **Sensor Screen:** this component displays a small screen below the operator's left hand with status about the selected robot's sensors if any.

All the aforementioned components will be discussed in detail in the following subsections.

5.1 Menu

The menu is by far the trickiest element of this interface since it requires all the components to be seamlessly intertwined. For that matter, it is necessary to break it down completely, from its integration in the editor to each of the button and switch's actions.

The menu, showed in Figure 22, was designed with a clear inspiration from the menus found in many VR games. This specific style of menu which is brought up in front of the user and sort of floats in the air finds its inspiration in VRChat's menu (Figure 23). VRChat is a popular VR application with millions of users, thus supporting its use as a design reference for this work's VR interface.

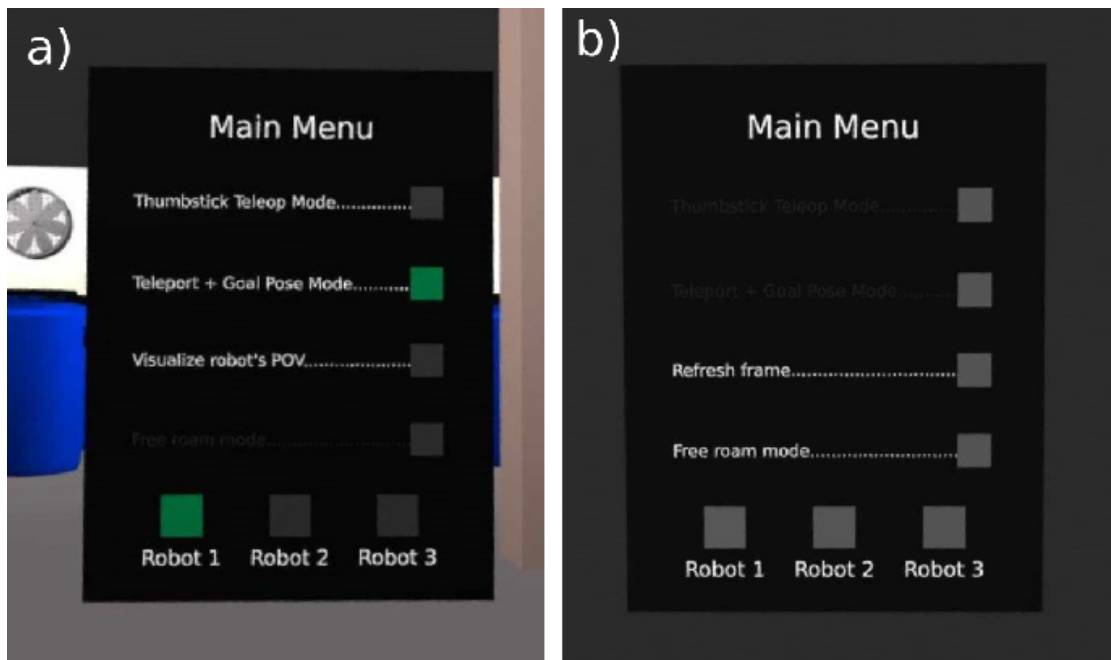


Figure 22. The interface's floating menu **a)** in the Free Roam mode and **b)** in the Robot POV mode.



Figure 23. VRChat's floating menu.

The Menu Bring-Up component, which opens the menu when a physical button is pressed on the VR controller, is attached to a "menu" empty game object, which is the root of sub-objects for each of the menu's buttons. The buttons themselves are "empty" objects, root of button meshes and labels. The Button components are attached to the button root objects. The same principle applies for switches. The Button and Switch handlers are attached to the "menu" object as well. The Figure 24 shows the difference in behavior between Switch and Button components and their respective handlers. Both components rely on the Emitter, a default WLE component which allows to convey messages between components. The "emitting" component has an emitter instance as class attribute, broadcasts messages with the `emitter.notify()` method while the "receiving" component has a callback of the emitter instance and receives messages through the `emitter.add()` method. The process is illustrated by the Figure 25.

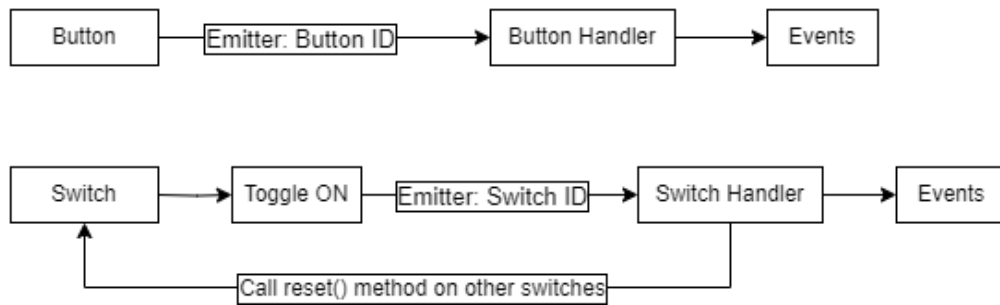


Figure 24. Comparison between the Button and Switch components.

The buttons act as momentary switches which are pressed once and released. In this interface they are used to select between the two distinct visualization modes:

- Third person view and free exploration around the scene, referred to as Free Roam mode.
- Inspection of the environment through the robot's perspective, referred to as Robot POV mode.

Pressing one mode button or the other activates the associated features, and deactivates those of the opposite mode. For the transition between modes to be clear and intuitive, the menu labels and behavior are updated dynamically. Upon choosing a mode, the text displayed on the menu changes to indicate which operations are available or not for the operator, and some buttons and switches are disabled. For instance, when entering the Robot POV mode, the user cannot enable robot teleoperation or goal pose. The switches are disabled, hovering the pointer above them does not generate any haptic feedback and the label of each mode is displayed with a darker font. Similarly, the Free Roam button is hidden when the operator is already exploring the scene. Finally, the "Visualize Robot POV" label which is displayed in the menu from the exploration

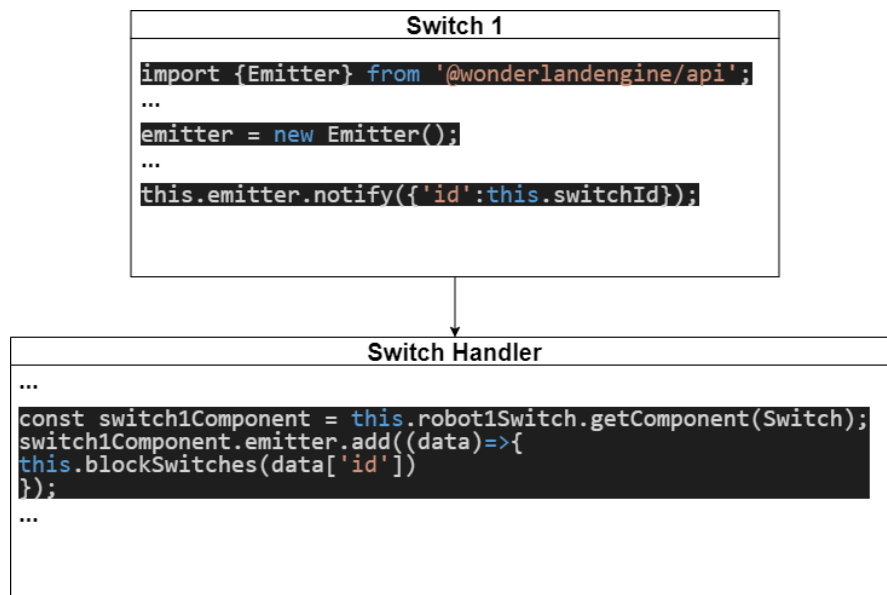


Figure 25. Relation between Switch and Switch Handler with the Emitter component.

- Free Roam - mode, turns into a "Refresh frame" label when the operator is currently visualizing the camera frames. The set of events following a button press on the Robot POV mode are broken down into a flowchart in Figure 26, while those of the Free Roam mode are illustrated in Figure 27. The Figure 22 displays the menu in the two control modes with its labels dynamically updated.

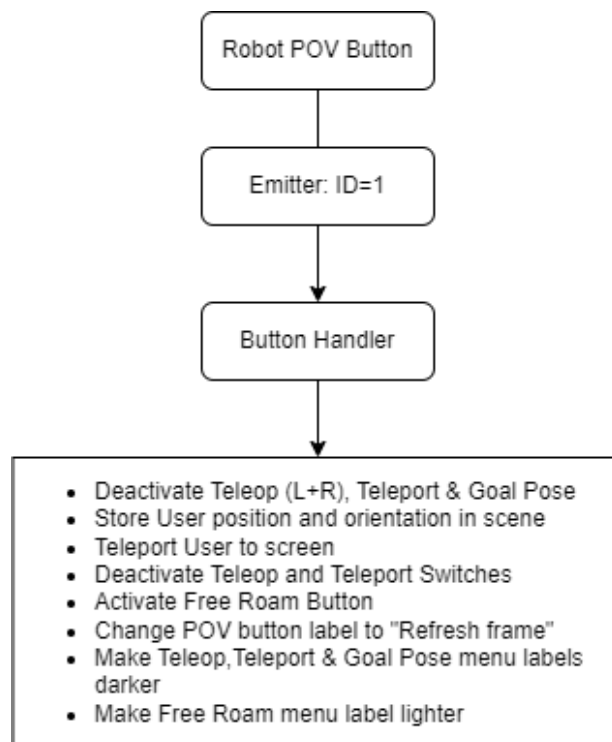


Figure 26. Events following a Robot POV mode button press.

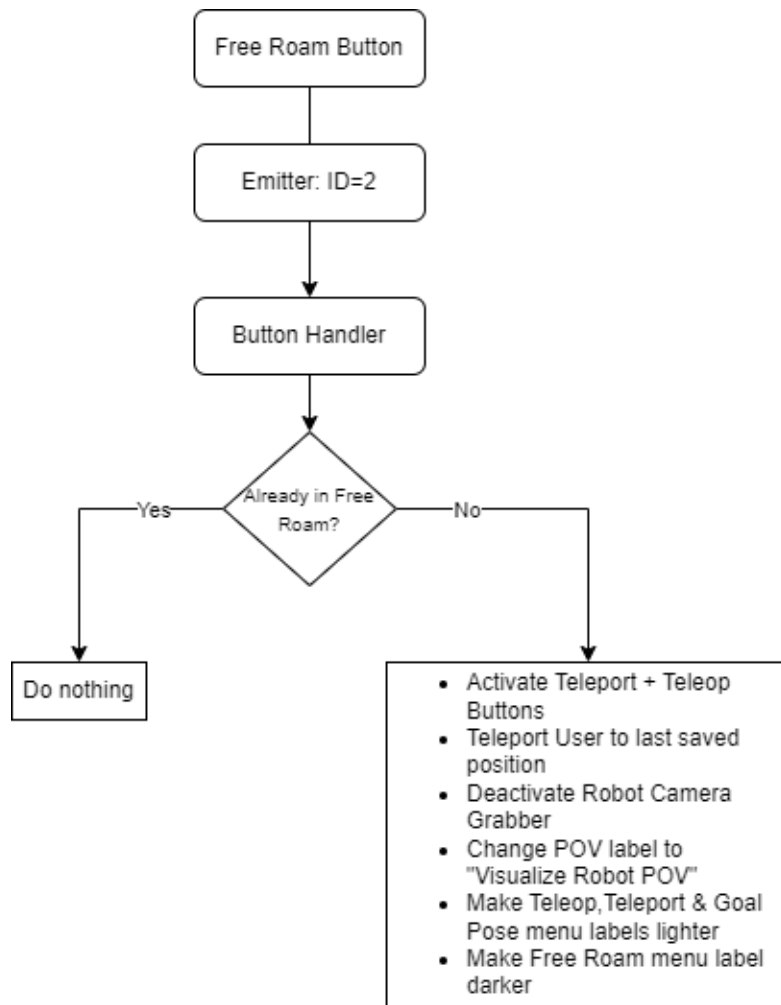


Figure 27. Events following a Free Roam mode button press.

After the buttons, the switches play a key role into this interface since they either select which robot should be focused on, but also the type of control the user should have on the selected robot. The menu counts five switches for a three-robot setup (Figure 22):

- Robot1 Switch.
- Robot2 Switch.
- Robot3 Switch.
- Teleop Switch, which activate the teleoperation components on both controllers.
- Teleport Switch, which activates the user teleportation on the right controller, and the goal pose pointer on the left controller.

Unlike buttons, switches stay engaged until another is pressed. While buttons and switches are equally effective for component interactions, switches offer a clear visual cue of their state: they change color and maintain it until they are reset, clearly indicating the active mode and selected robot. If the Robot 1 switch is engaged for instance, toggling the Robot 2 switch will reset the first automatically. The Switch class has a `reset()` method which resets the switch's appearance and the state of a "toggled" boolean to `False`. Figure 28 breaks down the effects of the Robot1 Switch, which are exactly the same for the two other robots, as well as the Teleop and Teleport switches. Only the emitted switch ID number changes, this ID ranges from 1 to 5 for a three-robot setup. Also, all switches are reset when switching between the Free Roam and Robot POV modes. This ensures the operator does not accidentally use the wrong control mode on the robots after spending some time visualizing robot camera frames.

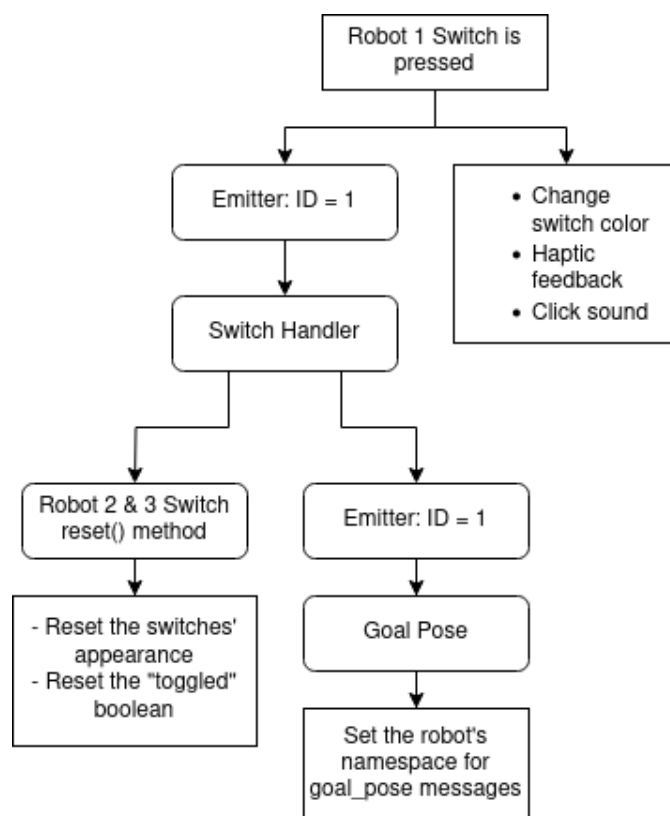


Figure 28. Events following a Robot1 switch press.

5.2 Sensor Screen

The Sensor Screen component displays a small screen below the operator's left hand. This screen conveys the current status of the selected robot's sensors. By pressing a robot selection switch, the sensor status screen displays the current robot that is selected, the type of sensor the robot is equipped with, and the value measured by the sensor. A threshold value can be set in the robot's parameters and if it is reached, a "WARNING!" red message is displayed on the screen. This warning message indicates that the operator should investigate by requesting a snapshot of the robot's surroundings for instance. The component receives the switch's ID from the Switch Handler's Emitter component. A method called `updateValues()` refreshes the screen's information.



Figure 29. The sensor screen showing information for Robot 1 when it is selected on the menu.

5.3 Robot Component

Each robot object in the scene has a mesh, which provides a 3D visual representation of the robot, and a Robot component which updates the robot's position with respect to the world at each frame.

5.4 Teleoperation, Goal Pose, Teleportation and Snap Turn

The Teleoperation, Goal pose, Teleportation and Snap Turn components are attached to the left and right "cursors" in the VR-rig hierarchy tree in WLE, to receive the operator's input on the VR controllers.

In traditional VR experiences, there are two methods of locomotion for the users: **Teleportation** and **Smooth locomotion**. The first mode allows the user to point somewhere, and their avatar is teleported to the end of the trajectory, usually represented by a parabola on the screen or a flat marker. This method is often preferred by novice VR users as it avoids the common feeling of discomfort - motion sickness - triggered by the second mode. Smooth locomotion is particularly deranging for beginners as the avatar performs a linear - smooth - motion within the 3D virtual environment, but the user is not moving in reality. The only remedy for VR-induced motion-sickness is usually to end the VR session, or practice more regularly to build a tolerance.

Since the accessibility of the interface is a requirement, teleportation was chosen. The operator of the robot fleet should be able to work extensive sessions without having to rest from motion sickness. Also, the Snap Turn component allows the operator to rotate their point of view by fixed angle increments in the VR scene, without having to physically turn their head around. This is performed by pushing the right controller's thumbstick to the left or right. Snap Turn is a very common comfort feature in VR applications, especially for those who use VR while seated.

The Goal Pose component is actually a modified version of the Teleportation component. It works the same way as the Teleportation component, to the difference that it does not teleport the user to the pointed location, but rather the coordinates are sent to the selected robot.

The Teleoperation component takes the same logic as the two previous components when it comes to user input, the thumbsticks' orientation is monitored: pushing the thumbstick forward and backwards moves the robot linearly, and pushing the thumbstick to the sides rotates the robot. Both the left and right thumbsticks produce the same output to ensure accessibility for left and right-handed operators.

All these components make use of the thumbsticks, to improve the user experience a dead-zone was introduced. It is a threshold past which thumbstick movements are registered. Without this dead-zone the operator would likely execute a "snap turn" command when pointing at a teleportation target. As a matter of fact, in practice it is difficult to push the thumbstick forward perfectly centered. The threshold filters any unwanted rotation. This is also very useful to make the robot's motion smoother when teleoperating.

5.5 Screen & Robot Camera Grabber Components

The Screen and Robot Camera Grabber components are attached to the display object which contains a screen mesh in the shape of an hemisphere to display wide-angle images. The Robot Camera Grabber is activated when the operator enters the Robot perspective mode, it waits for a robot to be selected. Once the switch is toggled the component sets the robot's namespace from the switch's ID, and sends a "snapshot" command via the Emitter to the Screen component (see Figure 30). The camera frame request is issued by the Screen component as shown in Figure 31.

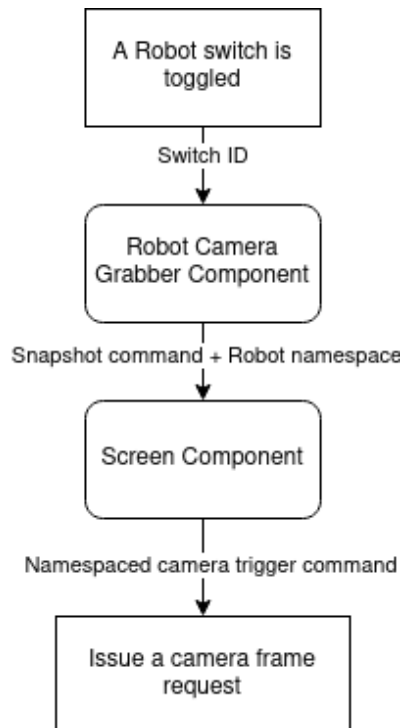


Figure 30. Snapshot command process.

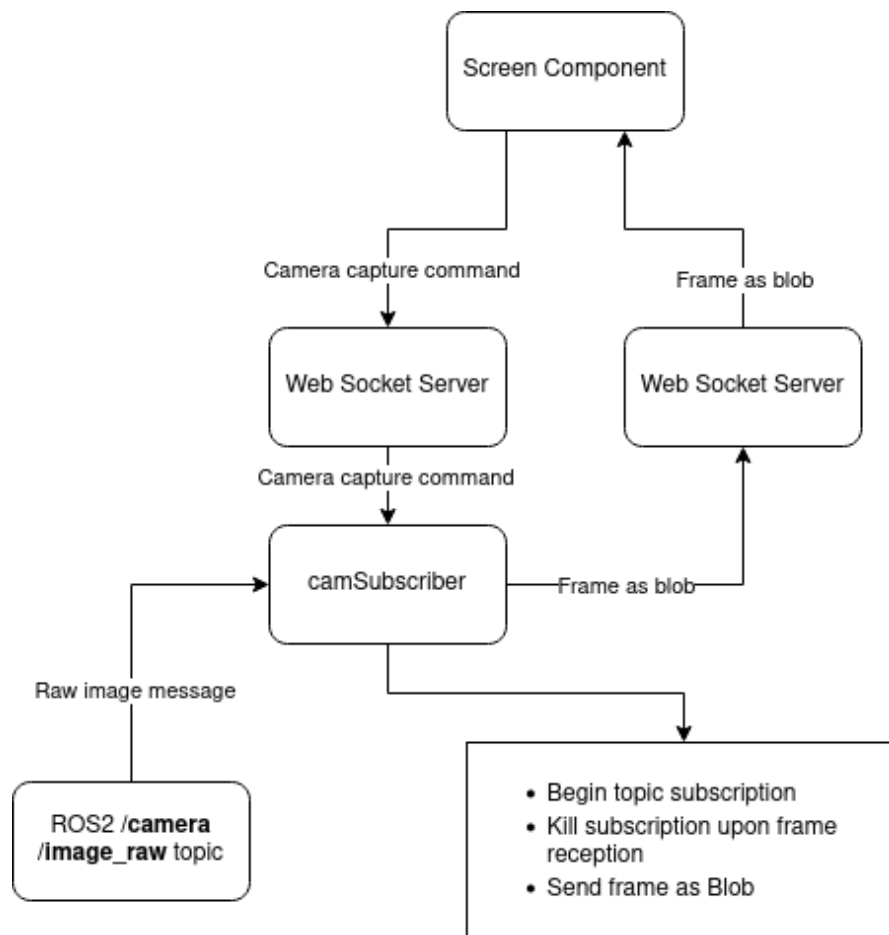


Figure 31. Camera frame request process.

5.6 Switching from VR to robot POV

After the menu, displaying camera frames into the interface was a great challenge. The point of this mode is to provide the operator with a visual of the robot's actual surroundings. Although the VR environment is a 1:1 replica of the remote one, in practice the robot could face an obstacle not shown in VR, or the operator might simply want to see through the "eyes" of the robot to better assess the situation. Instead of streaming a live feed of the robot's camera, this mode takes a single snapshot from the robot currently selected in the menu.

Displaying the images in VR Receiving images from the robots was challenging but having an already solid basis with the server's architecture, it was not the most troublesome aspect of this feature. The real struggle was actually displaying the image within the VR experience. To display an image in WLE, it needs to be applied to a material as a texture, and the material is attached to a mesh object. As suggested in the section about the engine, some materials are installed by default and can be modified from the editor (color, shadows, reflections, shader...), but they are usually common to several meshes in the scene, so changes apply to all the objects. The engine is designed to optimize the application build so it runs smoothly from the browser. Instead of storing multiple separate textures in the application files, it appends all the textures to a same atlas at startup. The issue with that approach is that every time a camera frame is received and applied as a texture, a back-process appends the newly received texture to the atlas. If the texture is refreshed at a high frequency, the entire VR experience jitters. This is why the interface only displays a single frame per request and not a live camera feed. Displaying several frames consecutively interrupts any kind of interaction from the user, like button presses to open the menu, and eventually the texture atlas cannot keep up and throws an error.

The screen used to display the camera frames in the VR interface is hemispherical. It was made with Blender from a hollow sphere cut in half. The surface was flipped in the model editing software because the texture must be applied inside the hemisphere and not outside. This "curved screen" shape was chosen to provide a more immersive visualization of the robot's POV with wide-angle images. As a matter of fact, flat images displayed on a flat screen in VR are not immersive: their field of view is limited and some of the robot's surroundings are not visible. Also, flat images do not convey any sense of depth. Depth is crucial for the operator's situational awareness [23], and as much information as possible should be visible on a single frame, without having to move the robot or rotate its camera. The solution is to use full-field panospheric images which can be captured with wide-angle and 360 degrees cameras, then displayed in VR with spherical or hemispherical displays [23]. The Figure 32 shows the process of applying a texture to the screen hemisphere using Blender's UV-Map editor. The image in this

preview comes from a simulated wide-angle camera (Figure 37). It allowed to accurately place the texture on the UV-Map. The orange element on the UV-map editor shows how the texture is wrapped on the geometry, presently an inside-out hemisphere. Once the 3D model is exported from Blender as **.gltf** extension, the model is attached a custom material to which textures can be applied from WLE, and the textures are automatically displayed correctly, as shown in Figure 33. However, this was by setting the texture from the editor, applying it in the interface at runtime is a complex process.

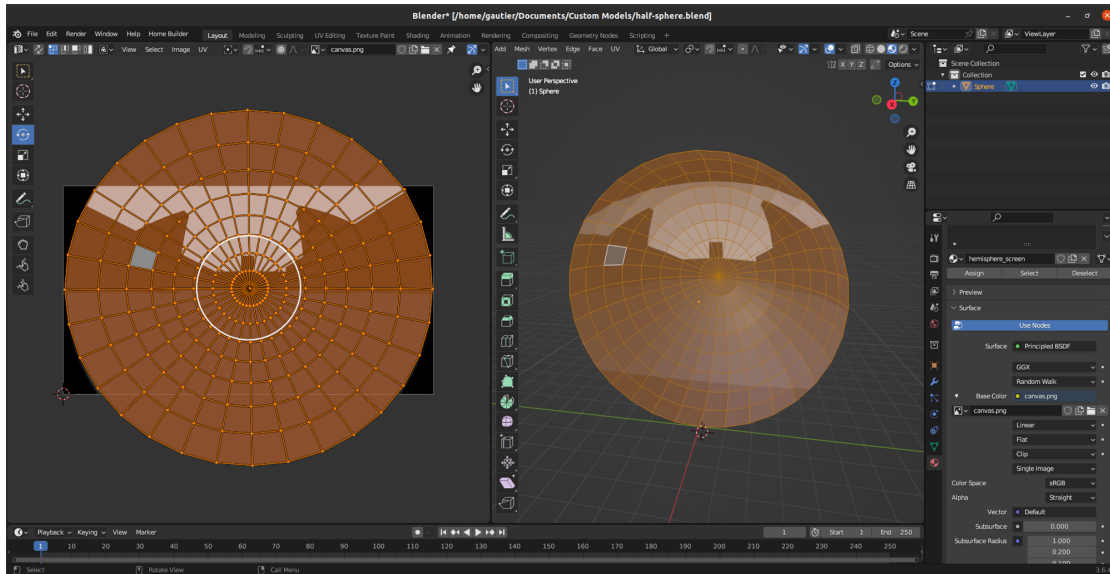


Figure 32. Preparing the hemisphere screen for wide-angle images with Blender's UV-Map editor.

Converting the camera frame to texture The way the Screen component issues camera frame requests was covered earlier, illustrated by Figure 31. However, the component is more complex than that, since it hosts the code responsible for the texture update. Once the component is active, it connects to the WebSocket server and emits camera frame requests.

The received message, a Blob, contains an RGB8 Uint8Array of size 2764800 (The camera is set to output at a resolution of 1280x720 pixels. The encoding of the images is RGB8, each color channel is encoded with a byte, there are three color channels in RGB, which makes 3 bytes per pixel. The step - representing the number of bytes per row - is 3840. $3840/3 = 1280$ pixels; $2764800/3840 = 720$ pixels.).

The WLE API for textures suggests to use "canvas" which are elements used to draw graphics on HTML pages using JavaScript, usually, images from web sources can be

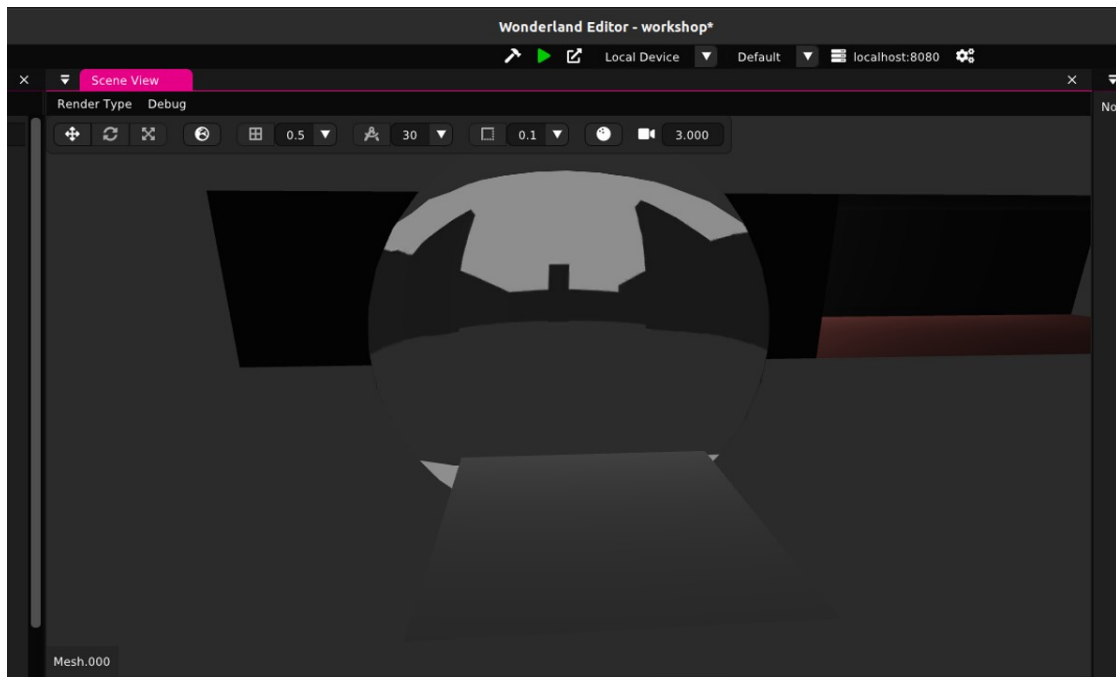


Figure 33. Wonderland Engine’s editor with the hemispheric screen and a camera frame applied as texture.

applied with their URL to a canvas, and within a 2D context on the web page, the canvas is displayed, containing the image. In the case of the interface, the image is received as an array which needs to be converted to the ImageData format before it can be applied to the canvas. Then, once the canvas is built, the WLE API comes into play: a texture object of the Texture class is initiated, attached to the Screen component’s material, itself attached to the Screen component’s mesh, presently the hemispheric screen. The texture object gets its texture attribute updated with the current canvas. The update() method for Texture classes appends the texture on the atlas slot. This complicated process is summarized in Figure 34.

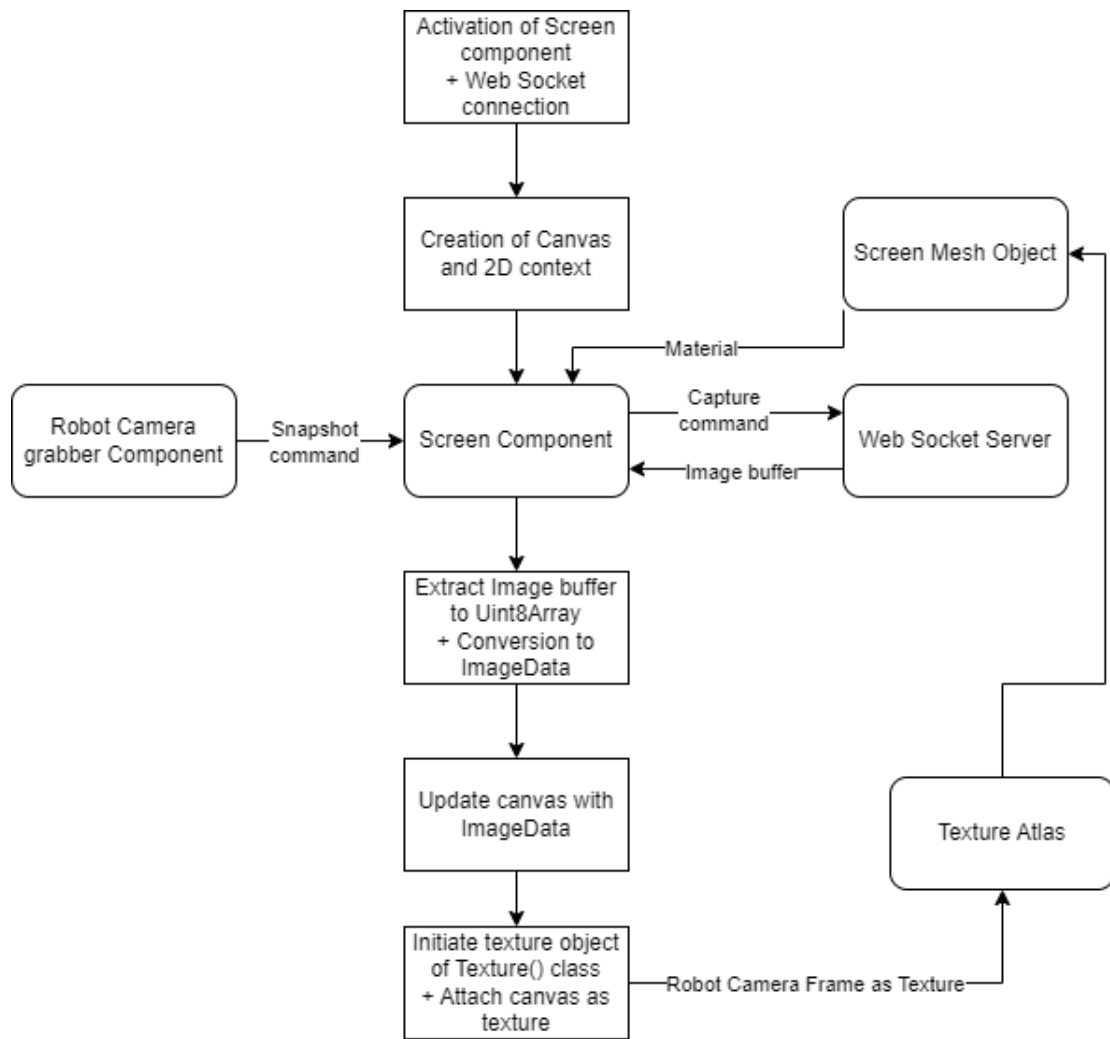


Figure 34. Summary of the conversion of camera frames to textures.

6 Demonstration

The demonstration serves as a proof of concept, showcasing the viability of the interface and its features in a simulated scenario introduced in the section **1.1 Problem Statement**. For this demonstration, the user plays the role of an operator tasked with the supervision of three inspection robots, in a simulated industrial facility with varied environments.

This section introduces the hardware and software elements involved in the demonstration, as well as the simulated environment and visuals from the operator's point of view.

6.1 Hardware setup

6.1.1 Oculus Quest 2

The headset at a glance. The Oculus Quest 2 [24] (Figure 35), is a 6-DOF Virtual Reality headset capable of performing room-scaling and hand tracking. By default, it is controlled with a pair of Touch controllers tracked by the headset's array of infra-red cameras, and those controllers can provide some haptic feedback to the user for more immersion. The headset operates independently, hosting games on an Android-based OS with its Qualcomm Snapdragon XR2 processor, or it can stream content from a VR-ready computer where the PC handles all the graphics processing with its high-end CPU and GPU. The Quest 2 is a reference in the realm of VR devices, breaking sales records to this day.

Setting up the headset. In order to install applications on the Quest 2 other than the official applications (from the Oculus Store), the headset needs to be set up for Android Debug Bridge (ADB), a development tool for Android debugging and sending commands to devices running the OS. Those commands also work wirelessly, one convenient way of activating **wireless** ADB support is by installing the SideQuest software on the host PC [25], after enabling the developer mode in the headset [26] which by itself enables ADB commands via USB.

6.1.2 Lenovo Ideapad 5 Notebook

The computer used throughout this project as a server is a simple Lenovo Ideapad 5 notebook, powered by an 11th generation Intel i7 processor, with 16GB of DDR4 RAM, 1TB of M.2 SSD storage, and Intel Iris XE. integrated graphics. The laptop has Ubuntu 20.04 installed as a dual-boot OS.

This computer was powerful enough to generate the 3D models for this project, and can manage to run the Gazebo simulation alongside the WebSocket server and the



Figure 35. The Quest 2 and its Touch controllers.

HTML server for the WebXR app. However, it is not "VR-ready" which means that it is not usable in a PCVR configuration. The aforementioned Oculus Quest 2 was used as standalone for the demonstration.

6.2 Software setup

6.2.1 Gazebo simulation

Although this project is carried out with hardware robots in mind, this demonstration involves a simulated robot fleet in a simulated environment in Gazebo 11, part of the ROS 2 robot middleware.

Mapping the environment Mapping the simulated environment is necessary to allow autonomous navigation for the simulated robot fleet. As it is a very common method in robotics, and this thesis is not aimed at evaluating navigation solutions but rather focus on the human-robot interface, off-the-shelf SLAM was used to generate a map with a single robot and its LiDAR. The documentation on the Turtlebot3 website contains a comprehensive tutorial about SLAM with simulated robots. By driving the robot at a slow speed around the simulated room, the Slam-Tools package nodes built a 2D map from the robot's LiDAR hitting the walls, which was saved and reused for autonomous navigation. Again, the robot's documentation came in handy since a full tutorial explains how to start the set of nodes necessary for navigation with the generated map. SLAM allows the robot to localize itself in the environment by identifying landmarks in the scene with its LiDAR, but if an unknown obstacle is sensed (a piece of furniture is added after the map was generated, or a person stands in front of the robot) the robot's path planner can adjust the trajectory to avoid it. At all times the robot's estimated position

and orientation within the map are published to ROS 2 topics thanks to the odometry. Figure 36 shows Gazebo and Rviz side by side as the map is loaded and SLAM is running. Rviz displays the robots, the map and the costmap. The latter shows the area where the robot can navigate freely and the walls and obstacles, with a buffer region around them (purple and blue), that the robot should keep away from.

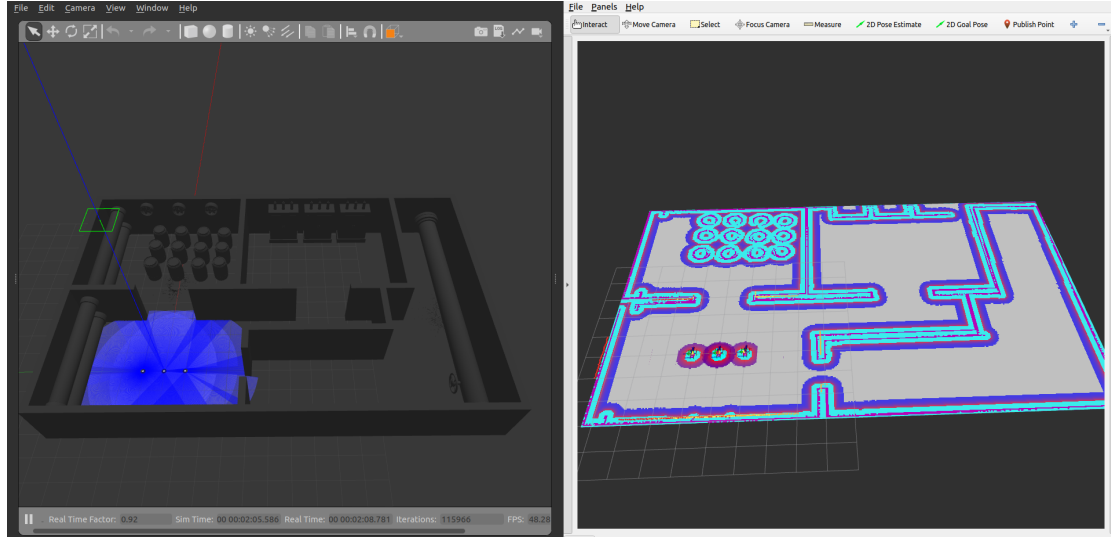


Figure 36. Turtlebot3 Waffle robots performing SLAM (Gazebo on the left, Rviz on the right)

Multi-Robot setup Once the SLAM navigation worked for a single robot, it was extended to support a fleet of several Turtlebot3 robots. Precisely, the simulation was constituted of three instances of the Waffle robot. Each of the simulated robots is loaded from a URDF file which corresponds to the characteristics of the real physical robot, and the set of nodes and topics generated by Gazebo are the same ones that would be involved in a physical Waffle robot fleet (Figure 18).

In the context of this thesis, the multiple-robot SLAM navigation was performed using the Nav2 navigation stack and the AMCL localization algorithm. Each robot is identical but has a namespace, which means each robot has the same set of ROS 2 topics and nodes which convey a plethora of information such as transforms, joints, camera frames, etc. If the default odometry topic for a single robot is `/odom` in the case of a multiple robot setup, the first robot's odometry would then be `/waffle1/odom`. The three robots are localized on the same map and can navigate autonomously to goal coordinates, while performing obstacle avoidance.

Simulated Cameras The real Waffle robot natively carries a camera, and so does its simulated counterpart. This camera is necessary to convey images of the robot's surrounding within the interface. However, the default simulated camera produces standard flat images as opposed to the wide-angle panospheric images that the interface expects.

It is possible to change the simulated robot's camera from its model properties. In the **turtlebot3_gazebo** ROS 2 package, a folder named **/models** contains all Turtlebot3 models from the Burger to the WafflePi. Inside the Waffle's folder the "model.sdf" file contains all the links and joints of the robot, with associated meshes for 3D representation, and ROS plugins for topics (odometry, camera...). Gazebo provides a tutorial to change the camera properties within the model configuration file. Both "fish-eye" and 360 cameras can be set up, however, simulated 360 images look quite bad. Wide-angle images are quite decent on the other hand, as shown in Figure 37.

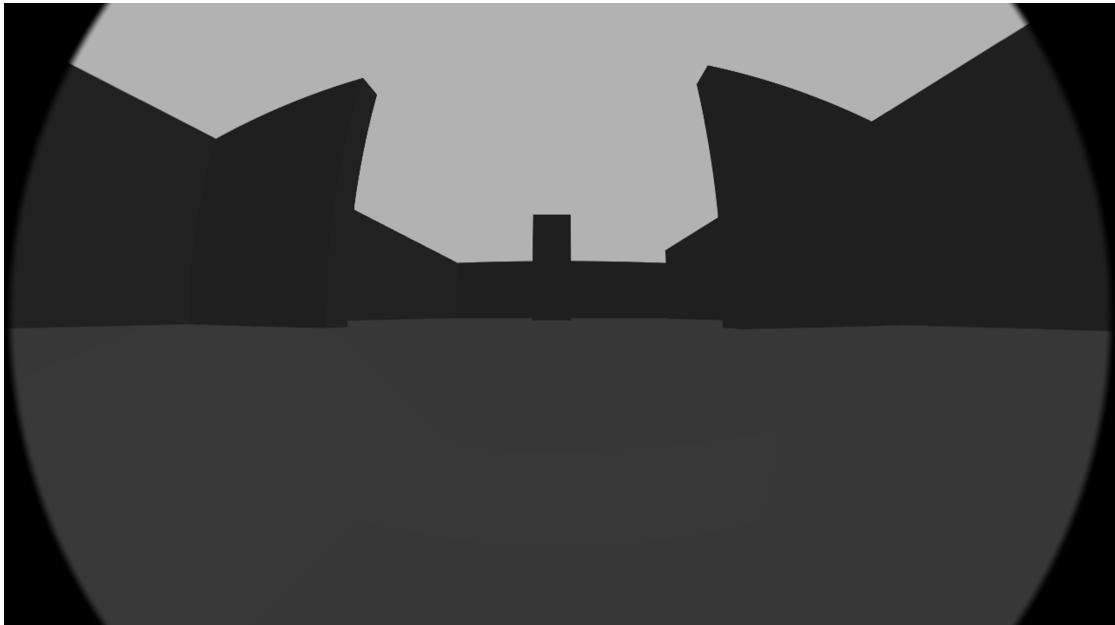


Figure 37. A wide-angle image from the TB3 Waffle's simulated fish-eye camera.

6.3 Demonstration in Simulated Environment

To test the interface, a simulated environment was modeled in line with the scenario introduced in 1.1. The scene was made in Blender like most of the 3D assets of the interface, using free 3D assets from the internet. The Figure 38 shows a preview of the demonstration scene in Blender. The demonstration scene contains the three inspection tasks from the scenario:

The first room - the chemical storage area - (Figure 39a) contains barrels filled with chemicals. One of them is leaking and a spill is visible on the floor. When approaching the spill, the first robot sends warning signals to the operator on the sensor status screen. The operator can request a visual of the robot's camera and the spill is visible on the picture. The operator's POV in VR is illustrated by Figure 40a and the robot's POV is shown in Figure 41a.

The second room - power distribution unit - (Figure 39b) contains three high-voltage electrical panels. The operator accesses the robot's thermal camera feed and does not detect any abnormal temperature looking at the panels. The temperature on the sensor status screen is also normal. The operator's POV in VR is illustrated by Figure 40b and the robot's POV is shown in Figure 41b.

The third and final room - the plant's plumbing room - (Figure 39c) contains a leaking spoiled water exhaust pipe. The third robot detects high humidity levels and a visual of the surroundings reveals a puddle underneath the pipe. The sensor status screen reports high humidity levels. The operator's POV in VR is illustrated by Figure 40c and the robot's POV is shown in Figure 41c.

For each task, the operator sends a goal position for each robot in the respective rooms they have to inspect, and observes warning signals in case of hazard.

The source code for reproducing this demonstration is available on GitHub¹.

¹https://github.com/Gautier30/ROS2_VR_Interface

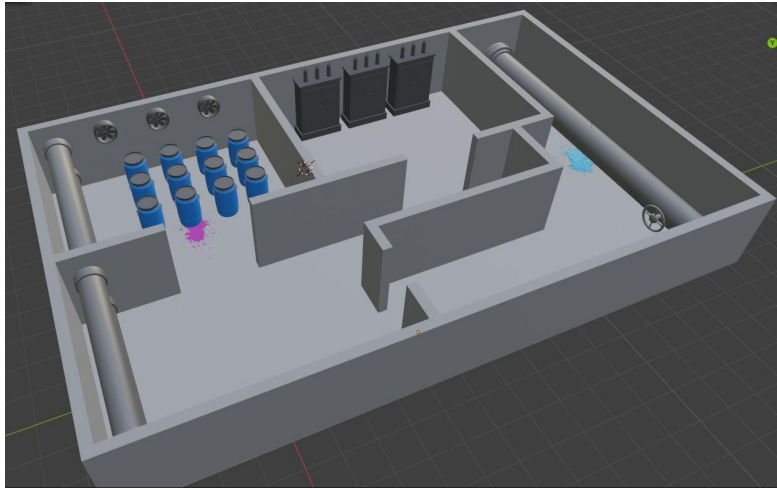


Figure 38. Blender preview of the demonstration scene.

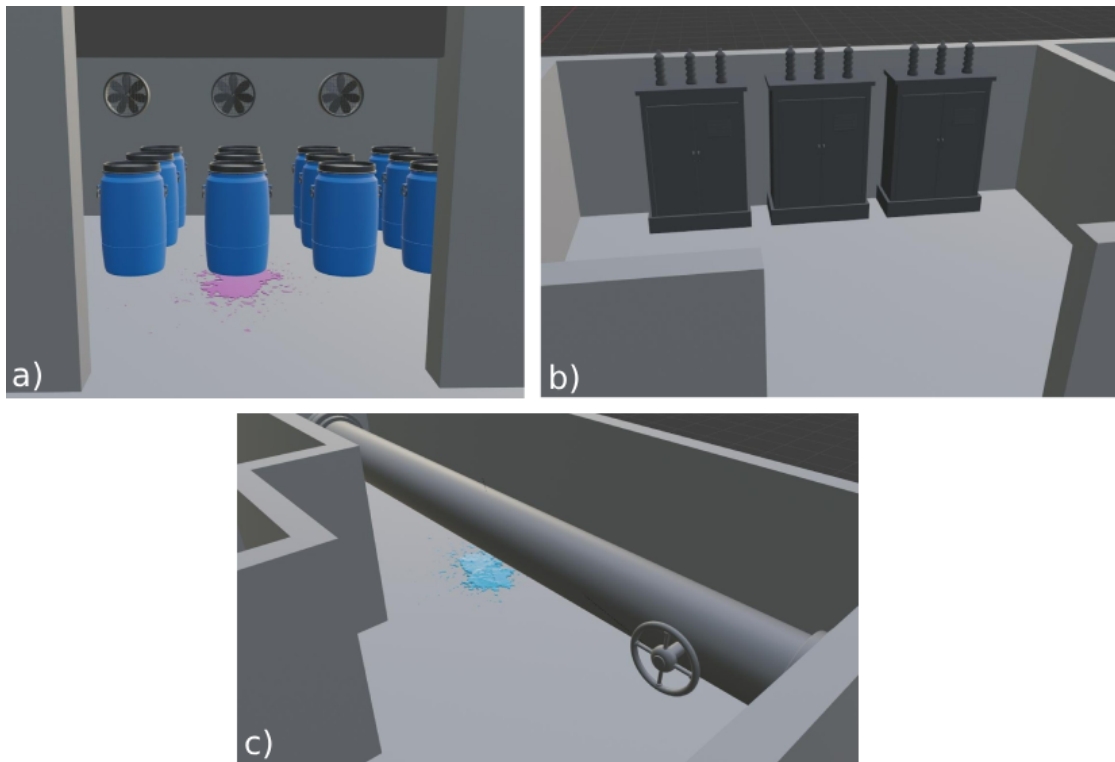


Figure 39. Close-up of the demonstration's three tasks: **a)** inspection of a chemical storage room, **b)** inspection of a power distribution room, **c)** inspection of a spoiled water exhaust pipe.

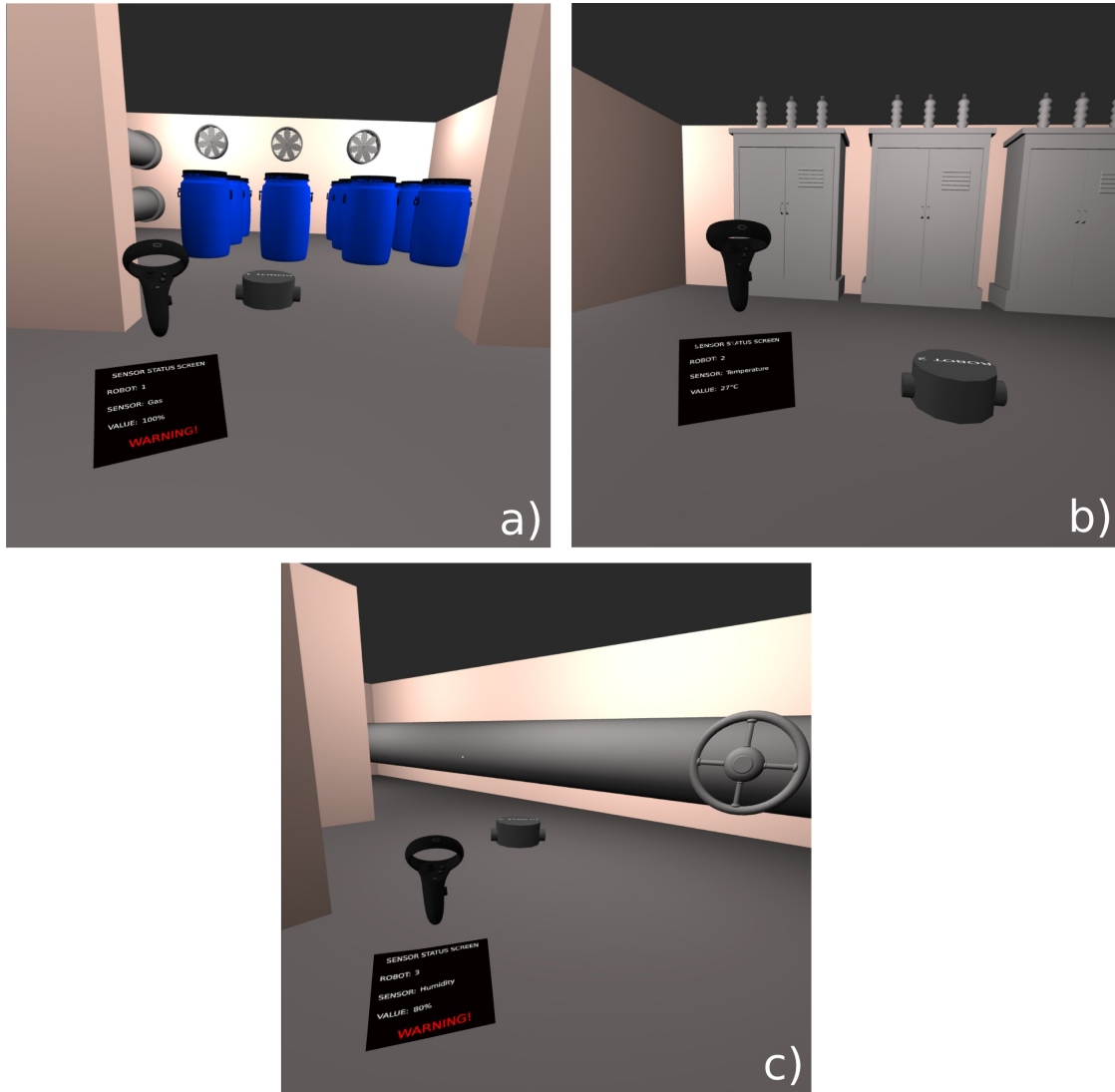


Figure 40. Operator's point of view in VR for the three tasks: **a)** inspection of a chemical storage room, **b)** inspection of a power distribution room, **c)** inspection of a spoiled water exhaust pipe.

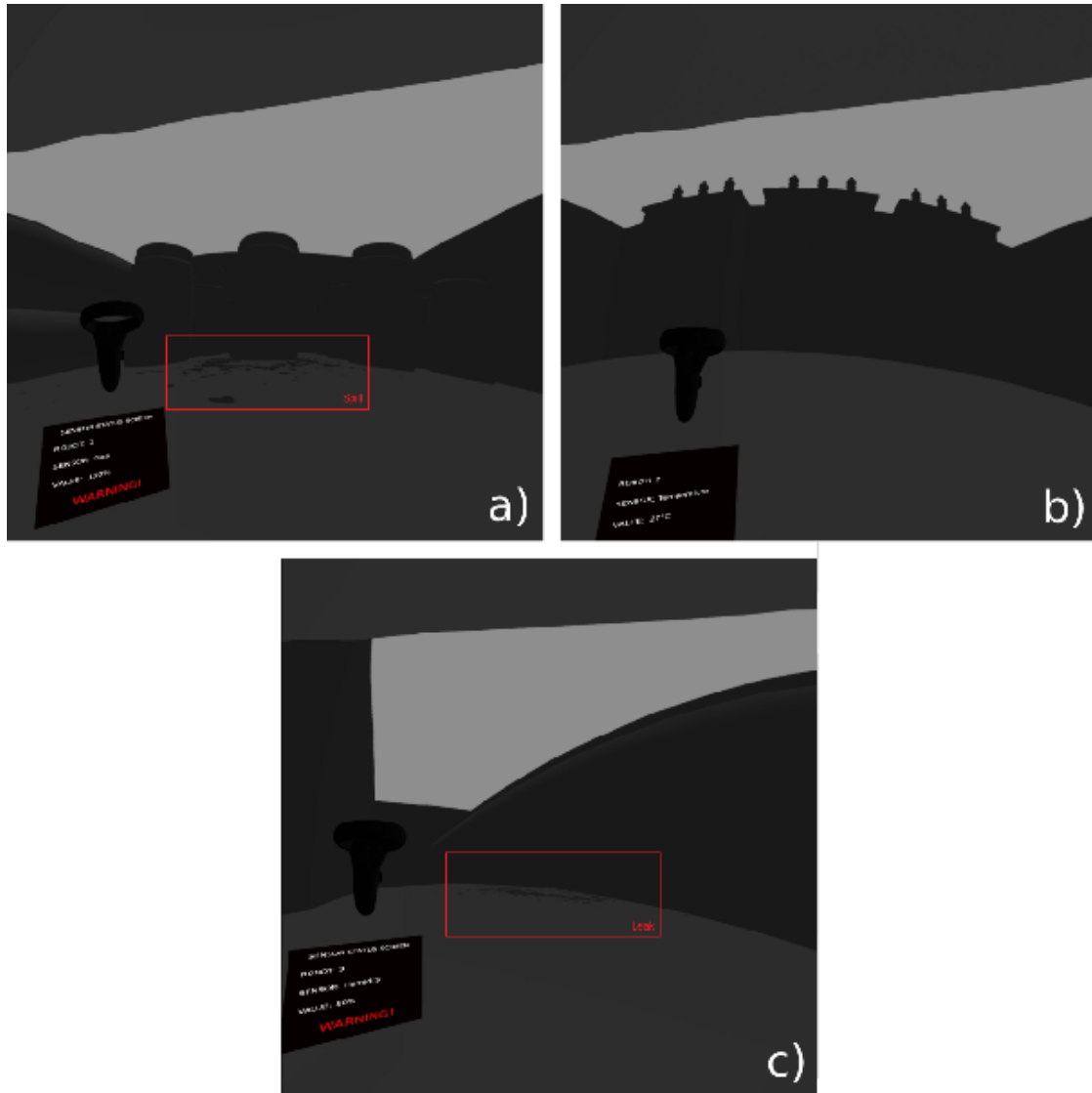


Figure 41. Robot's point of view for the three tasks: **a)** inspection of a chemical storage room, **b)** inspection of a power distribution room, **c)** inspection of a spoiled water exhaust pipe.

7 Discussion

The goal of this section is to go over the challenges encountered during this project, as well as covering the future work. The point is to provide the reader with an insight on what was attempted but did not work for the interface, and share tips and tricks that came in handy during the design process.

7.1 Hand Tracking

Although the project requirements include natural gestures for the operator interaction (Requirement 1), the current state of the demonstrated VR interface makes use of the VR controllers to issue goals for the robots and to teleoperate them. The Oculus Quest 2 is compatible with hand tracking, and this feature is natively supported by WebXR and by extension Wonderland Engine. Some experiments with the engine's hand tracking were conducted in a separate VR scene:

Project templates show various hand displays, with either a skin or the individual joints, and the hand tracking default scripts are quite straightforward to understand and alter. A simple "index pointing" detector method was added to the default hand tracking script, to track operator's pointing gestures. The method simply computes the distance between the middle finger and index finger tips, and if the distance is greater than a set threshold, then the hand is registered as pointing. The method is called by the teleportation script so that the operator can teleport across the VR scene not by pushing a thumbstick this time, but instead by pointing into a direction, as illustrated by Figure 42a. This could be implemented for the goal position script too.

Taking the inspiration from Meta's Horizon World social application menu (Figure 43), a wrist button concept was also implemented to bring the menu up. The operator would have to rotate their wrist to reveal the button, and press it with the other hand's index finger, as illustrated by Figure 42b. When the wrist is in a normal position the button is not visible. Meta's approach is to display the button when staring at the wrist regardless of its rotation. This wrist method is becoming a popular approach among recently released VR titles.

Although hand tracking is well integrated into the API, and the aforementioned examples look promising when it comes to delivering a gesture-based control. The transition between the hand tracking and the controllers was found to be quite unreliable. The operator should still be able to teleoperate the robots, with a degree of precision that thumbsticks satisfy, but in practice jumping back and forth between gesture control and thumbstick control broke the interface's menu and interaction.

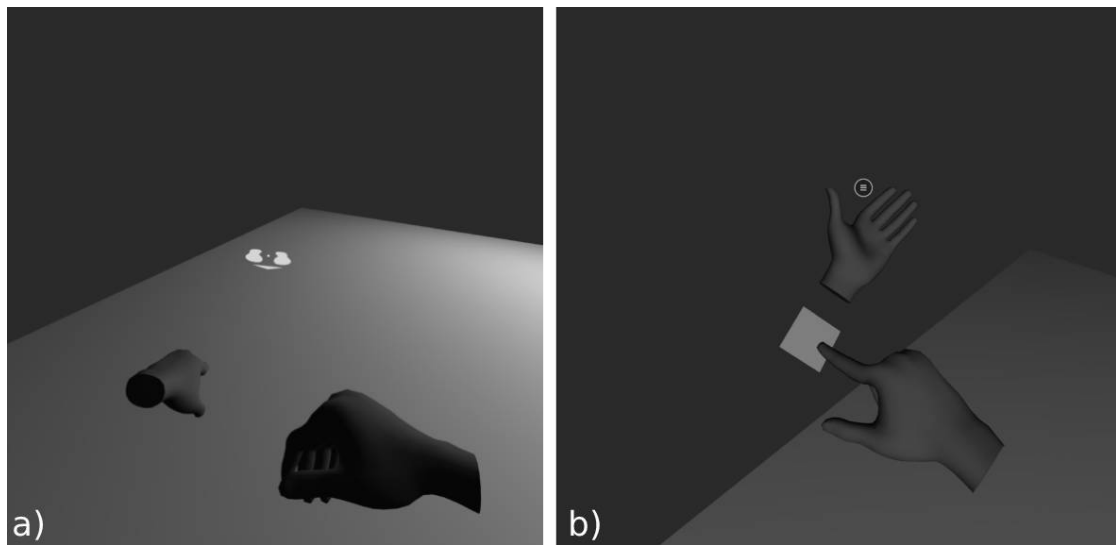


Figure 42. **a)** Teleportation based on pointing gestures. **b)** Wrist button concept to open the menu with hand tracking.

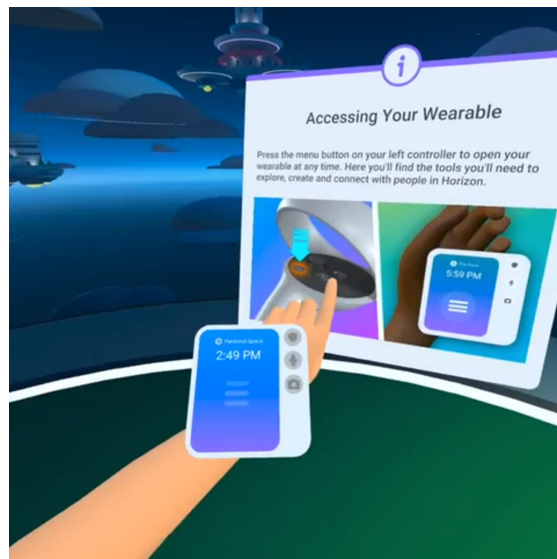


Figure 43. Meta's Horizon Worlds wrist "wearable" user menu.

7.2 Correcting SLAM generated maps

One nice trick that was useful when mapping a more intricate building: if the robot is driven too fast or takes sharp turns during mapping, the walls can be deformed in a way that two parallel walls will turn out crooked. But the output of the cartographer being an SVG image and a YAML configuration file, the image can be edited in any photo editing software, like Gimp, and the walls can be redrawn by hand, or artifacts can be erased. The map displayed in Figure 36 was actually crooked and fixed by hand. A before/after comparison is showed in Figure 44.

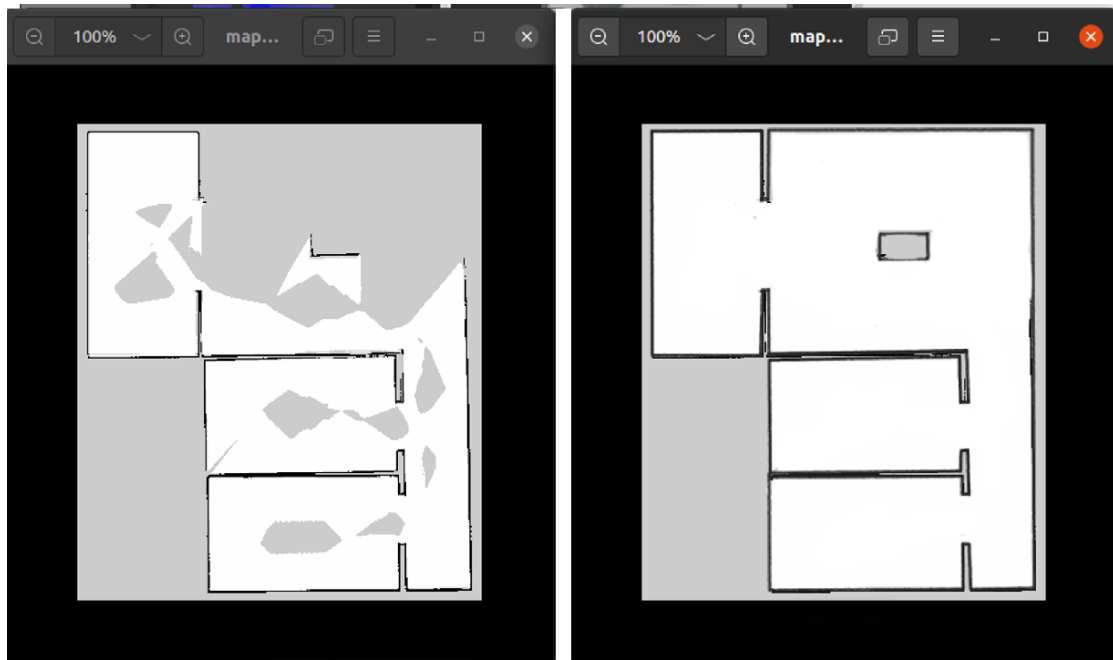


Figure 44. Comparison between a raw SLAM map and its edited version

7.3 Why not Unity?

Unity was the first game engine considered for this thesis as it is commonly used for robotic simulations, and also because one of this project's requirements is to leverage "off-the-shelf" solutions to ensure accessibility of this work. However, the integration of Unity with ROS 2 presented unique challenges, as discussed in this subsection, ultimately leading to the exploration of alternative solutions.

Unity at a glance Unity is a cross-platform game engine developed by Unity Technologies, it is used to make both 3D and 2D experiences for a plethora of devices such as

desktop computers, smartphones, consoles and it can also make VR applications for the Hololens 2 (Microsoft), PSVR (Sony Playstation), Quest (Meta, formerly Oculus)... The engine is written in C++, but the scripts are written in C# [27].

ROS 2 with Unity Unity is not natively compatible with ROS 2, but there exist a library that allows to integrate scripts such as Publishers and Subscribers to interact with ROS 2 topics: the library is called "ros2-for-unity". The official repository states that the compatible ROS 2 distributions are Galactic and Humble [28], however, the experiments conducted for this thesis proved that the basic features of the library worked with Foxy as well.

Unity for the VR interface As expected, it was quite straightforward to build a test world made of four walls, a mock-up robot model, and a cube for orientation reference, as shown in Figure 45.

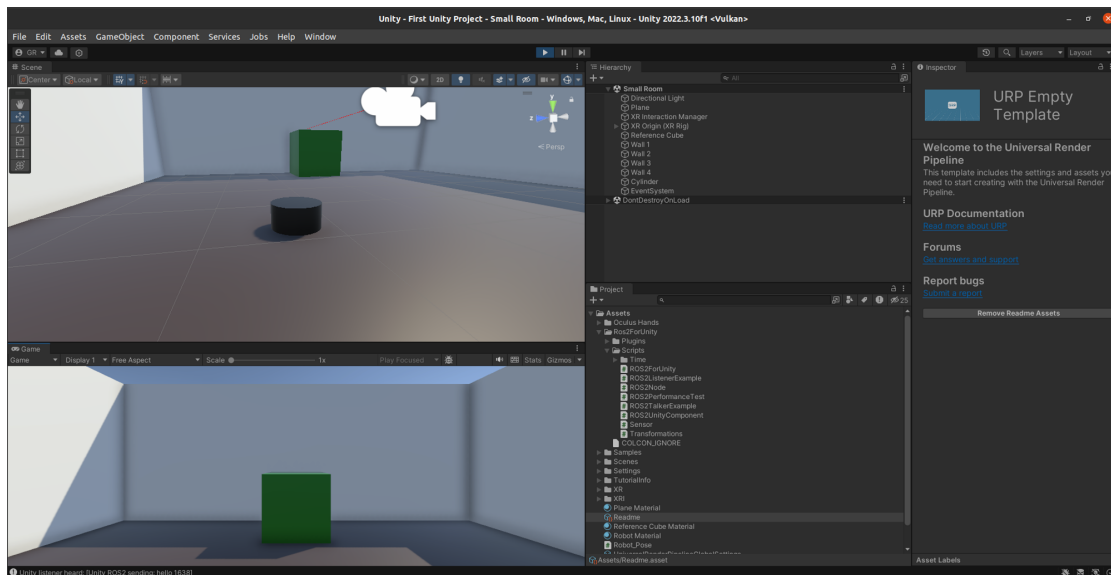


Figure 45. Sample scene in Unity Engine, editor view on top and VR render at the bottom

Unity being so popular, many tutorials can be found on the web, specifically the tutorial by the UnityRos2Tech channel on how to setup the ROS 2 library with Unity [29]. With a default ROS 2 library script to subscribe to the /tf topic, and a few changes to apply the transformations to the 3D model representing the robot, it was very straightforward to get the model to move according to a single simulated robot. That was for the preview render in the editor. However, the same should work from the VR headset. Again, relying on YouTube tutorials, it was fairly easy to build the VR application for Android and install it to the headset. Unfortunately, the application did not track the

simulated robot's movements. It comes down to a limitation of the "ros2-for-unity" library: the 3D simulation, or game, needs to run on the same machine which hosts the ROS 2 nodes and topics. The Quest is connected on the same local network but does not access the robot's information. For that reason, further development with Unity was interrupted. A PCVR setup is required for the Unity experience to work, however, such a complex setup is not in line with the requirement of the setup to be accessible.

Lastly, having to build the application after every code change, and having to install the APK all over again in the headset, was a very slow process. This aspect was incompatible with the project's last requirement being to favor development software capable of quickly deploying prototype iterations.

7.4 Live Video Stream

A prototype of video stream was implemented on the laptop's browser with a custom HTML file and some scripts, leveraging HTML canvas, and it gave quite satisfying results, but the same principle could not be replicated in WLE because of the way the API handles textures.

7.5 Future Work

The short time to complete this work led to some aspects needing to be explored. Several elements could be further investigated to improve upon the solution that this thesis delivers:

First, a proper user evaluation should be performed to provide a more comprehensive review of the interface's usability. The feedback of a single test-user allowed to refine the interface's user experience through several iterations, but a broader assessment is necessary to draw meaningful conclusions regarding accessibility and the enhancement of the operator's situational awareness. Those two criteria were listed as requirements of the interface.

Hand tracking is one of the accessibility features drafted but left unfinished, although it could significantly improve the user experience. Unfortunately, the current transition between controller and hand tracking that is proposed by the Oculus Quest 2 and the Wonderland Engine API is too unreliable and requires too much effort from the operator. This issue should be patched by the maintainers of the hardware and the API, or further focus on the interface's scripts could allow to smoothen the transition.

About the virtual environment, the one used for the demo was generated beforehand with a 3D model editing software (Blender). Still, the principle presented by [15], where the robot dynamically generates the 3D environment as it explores, could be integrated with this thesis' interface.

Also, the visualization of the robots' real environment is currently restricted to still images, which limits the possibilities in terms of teleoperation. An investigation is needed to determine if the Wonderland Engine is at capacity or if new updates and changes in the API could allow a real-time video stream.

The demonstration scenario presented in this thesis involves a homogeneous fleet of three Turtlebot3 Waffle robots, but the current features of the interface should be compatible with any robot. It would be interesting to test the interface with a mobile platform, loaded with a manipulator to close the valve in the third task of the demonstration inspection. This would use the teleoperation capability intensively, and one of the joysticks could be remapped to control the manipulator's end effector instead of the platform's movements.

Finally, the interface's current state provides situational awareness through visual and haptic feedback, however, sounds could be another effective source of immersion according to Bremner et al. [30] and their research on the impact of data sonification in VR robot teleoperation.

8 Conclusion

This thesis presented the development of Virtual Reality based user interface for remote inspection in hazardous environments via a multi-robot fleet. The research focused on enhancing teleoperation and monitoring of a semi-autonomous robot fleet through an immersive VR interface. The latter aims to improve operator efficiency and situational awareness. Key achievements include successfully developing a VR interface that allows direct and intuitive interaction with the fleet, and implementing a custom-built WebSocket server that enables communication between the ROS 2 robot middleware and the interface. The interface was made with Wonderland Engine, a new contender on the game engine market, which proved to be a significant development tool for deploying each iteration to the VR headset.

Despite various challenges, such as the complexity of integrating ROS 2 with an unsupported game engine, or ensuring real-time responsiveness when overlaying the virtual environment with pictures of the robot's cameras, the thesis provided a practical solution, demonstrating the feasibility and effectiveness of the proposed framework. The research findings contribute to the fields of robotic and VR, showcasing the potential of VR technologies in transforming the way we interact and control robotic systems.

The demonstration that accompanies this thesis serves as a proof of concept as to the viability of such a solution, however, the user experience could benefit from the future work described in the **Discussion** section.

Acknowledgments

I would like to express my gratitude to my supervisors, Robert Valner and Ulrich Norbistrath, for their guidance throughout this project. As my first academic endeavor, their regular and insightful feedback was crucial in shaping and efficiently organizing my work with the very little time I was given. Leveraging Robert Valner's expertise in Robotics and Ulrich Norbistrath's experience in computer graphics and Virtual Reality, I was able to undertake this exciting project and contribute to the field of robotic inspection.

Thanks to the Wonderland Engine Discord community for their tremendous help early on in the project when I was still learning how to use JavaScript. Their reactive and clear advice made the learning curve much more gentle than it looked at first glance.

I also wish to extend my appreciation to the the OpenAI team for Chat GPT 4 which I used throughout the project to optimize my time designing the framework and putting it into words. The large language model helped me by suggesting synonyms and style corrections to improve readability. It also provided boilerplate code for me to use as a basis for deeper feature development (ex: basic WebSocket client code in JavaScript). GPT was used also as an idea generator, prompted with some of my general, unstructured, ideas and criteria for the demonstration. Finally the AI helped me expand the literature collection which was used in the background work section of the thesis, thanks to the Bing search feature.

Finally, I would like to thank my family and friends for their trust and support since the beginning of my Engineering studies. More than my future profession, engineering is my passion but this field rhymes with challenges, stress and doubts, which my caring surroundings made more bearable.

Gawher Reynes


References

- [1] Jeffrey Delmerico et al. “The current state and future outlook of rescue robotics”. In: *Journal of Field Robotics* 36 (7 Oct. 2019), pp. 1171–1191. ISSN: 15564967. DOI: 10.1002/rob.21887.
- [2] H. Miura et al. “Plant inspection by using a ground vehicle and an aerial robot: lessons learned from plant disaster prevention challenge in world robot summit 2018”. In: *Advanced Robotics* 34 (2 Jan. 2020), pp. 104–118. ISSN: 15685535. DOI: 10.1080/01691864.2019.1690575.
- [3] V. Michal. “Remote operation and robotics technologies in nuclear decommissioning projects”. In: Elsevier, 2012, pp. 346–374. DOI: 10.1533/9780857095336.2.346.
- [4] Frank E Schneider and Dennis Wildermuth. *Assessing the Search and Rescue Domain as an Applied and Realistic Benchmark for Robotic Systems*.
- [5] G. J.M. Kruijff et al. “Designing, developing, and deploying systems to support human-robot teams in disaster response”. In: *Advanced Robotics* 28 (23 Dec. 2014), pp. 1547–1570. ISSN: 15685535. DOI: 10.1080/01691864.2014.985335.
- [6] M. Hutter et al. “ANYmal - toward legged robots for harsh environments”. In: *Advanced Robotics* 31 (17 Sept. 2017), pp. 918–931. ISSN: 15685535. DOI: 10.1080/01691864.2017.1378591.
- [7] Tomislav Horvat et al. “Inverse kinematics and reflex based controller for body-limb coordination of a salamander-like robot walking on uneven terrain”. In: Sept. 2015. DOI: 10.1109/IROS.2015.7353374.
- [8] Manolis Chiou et al. “Robot-Assisted Nuclear Disaster Response: Report and Insights from a Field Exercise”. In: (July 2022). URL: <http://arxiv.org/abs/2207.00648>.
- [9] Boris Gromov, Luca Maria Gambardella, and Alessandro Giusti. “Video: Landing a Drone with Pointing Gestures”. In: Mar. 2018, pp. 374–374. DOI: 10.1145/3173386.3177530.
- [10] Jiang Zainan et al. “Virtual Reality-based Teleoperation with Robustness Against Modeling Errors”. In: *Chinese Journal of Aeronautics* 22 (3 June 2009), pp. 325–333. ISSN: 1000-9361. DOI: 10.1016/S1000-9361(08)60106-5.
- [11] Peter Kazanzides et al. “Teleoperation and Visualization Interfaces for Remote Intervention in Space”. In: *Frontiers in Robotics and AI* 8 (Dec. 2021). ISSN: 22969144. DOI: 10.3389/frobt.2021.747917.
- [12] F. Pugin, P. Bucher, and P. Morel. “History of robotic surgery : From AESOP® and ZEUS® to da Vinci®”. In: *Journal of Visceral Surgery* 148 (5 Oct. 2011), e3–e8. ISSN: 1878-7886. DOI: 10.1016/J.JVISCURG.2011.04.007.

- [13] Marcos de la Cruz et al. “Preliminary work on a virtual reality interface for the guidance of underwater robots”. In: *Robotics* 9 (4 Dec. 2020), pp. 1–24. ISSN: 22186581. DOI: 10.3390/robotics9040081.
- [14] Pooya Adami et al. “Effectiveness of VR-based training on improving construction workers’ knowledge, skills, and safety behavior in robotic teleoperation”. In: *Advanced Engineering Informatics* 50 (Oct. 2021). ISSN: 14740346. DOI: 10.1016/j.aei.2021.101431.
- [15] Patrick Stotko et al. “A VR System for Immersive Teleoperation and Live Exploration with a Mobile Robot”. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 3630–3637. DOI: 10.1109/IROS40897.2019.8968598.
- [16] Krzysztof Adam Szczurek et al. “Mixed Reality Human-Robot Interface with Adaptive Communications Congestion Control for the Teleoperation of Mobile Redundant Manipulators in Hazardous Environments”. In: *IEEE Access* (2022). ISSN: 21693536. DOI: 10.1109/ACCESS.2022.3198984.
- [17] Meta. *The future of VR - top trends for 2023*. <https://www.workplace.com/blog/the-future-of-vr>.
- [18] *WebXR explainer*. <https://github.com/immersive-web/webxr/blob/master/explainer.md>.
- [19] *WebXR showcase*. <https://immersiveweb.dev/>.
- [20] *Wonderland Engine*. <https://wonderlandengine.com/about/what-is-wle/>.
- [21] John Houston. *ROS 2: The Transition from Research to Production*. <https://www.freshconsulting.com/insights/blog/ros-2-the-transition-from-research-to-production/>.
- [22] *RCLNodeJS, ROS2 Javascript client library*. <https://github.com/RobotWebTools/rclnodejs>.
- [23] Veiko Vunder et al. “Improved Situational Awareness in ROS Using Panospheric Vision and Virtual Reality”. In: *2018 11th International Conference on Human System Interaction (HSI)*. July 2018, pp. 471–477. DOI: 10.1109/HSI.2018.8431062.
- [24] Wikipedia Contributors. *Oculus Quest 2*. https://en.wikipedia.org/wiki/Quest_2.
- [25] Adafruit. *Install and Use Sidequest*. <https://learn.adafruit.com/sideload-on-oculus-quest/install-and-use-sidequest>.
- [26] Adafruit. *Enable developer mode*. <https://learn.adafruit.com/sideload-on-oculus-quest/enable-developer-mode>.

- [27] Wikipedia Contributors. *Unity*. [https://en.wikipedia.org/wiki/Unity_\(game_engine\)](https://en.wikipedia.org/wiki/Unity_(game_engine)).
- [28] *Ros2 for Unity*. <https://github.com/RobotecAI/ros2-for-unity>.
- [29] *Ros2 for Unity setup tutorial*. <https://www.youtube.com/watch?v=1X6uzrvNwCk>.
- [30] Paul Bremner, Thomas J. Mitchell, and Verity McIntosh. “The impact of data sonification in virtual reality robot teleoperation”. In: *Frontiers in Virtual Reality* 3 (2022). ISSN: 2673-4192. DOI: 10.3389/frvir.2022.904720. URL: <https://www.frontiersin.org/articles/10.3389/frvir.2022.904720>.

Appendix

Source Code

This thesis PDF comes in a .zip archive along with supplementary archives containing the source code of the interface and the ROS 2 workspace for the simulation.

Factory.zip contains the Wonderland Engine project. After installing the latest version of the engine from the official website: <https://wonderlandengine.com/downloads/>, the project can be loaded and executed on the HTML server. Alternatively, the deploy/ folder can be hosted directly on a separate HTML server following these instructions <https://wonderlandengine.com/release/>.

Note: the interface was tested on a local network and the IP address of the server is hard-coded in some of the JavaScript scripts under the Factory/js/ directory. You should set the IP of your server at the appropriate line in the following scripts: screen2.js, robot-component.js, goal-pose.js and robot-teleop.js.

The archive also contains the WebSocket server files under the Factory/server/ directory.

turtlebot3_ws.zip contains the ROS 2 workspace for the Gazebo simulation. To execute it, ROS 2 Foxy must be installed, as well as all the Turtlebot3 for ROS 2 Foxy dependencies (follow the guide here: <https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/>).

In the workspace, the launch files useful for the simulation are contained in the following directory:

turtlebot3_ws/src/turtlebot3_multi_robot_sim/launch/

Running the Interface

Note: The interface is experimental. Running it requires several terminals, and the order of execution of each part is important. If a terminal contains an error, close all processes with CTRL+C and start over.

To run the simulation and the interface, first launch the VR app on the HTML server (It is much easier from WLE). Then, start the WebSocket server with the start_term.bash script (This should open several terminals, check for errors). Then, run the simulation with (in order):

multi_robot_spawn_launch.xml
multi_robot_slam_launch.launch.py

GitHub Repository

The project's source code is also available on GitHub at https://github.com/Gautier30/ROS2_VR_Interface.

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Gautier Reynes**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

VR-Enhanced Remote Inspection Framework for Semi-Autonomous Robot Fleet,

(title of thesis)

supervised by Robert Valner and Ulrich Norbistrath.
(supervisors' names)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Gautier Reynes
09/11/2023